

25
ADVANCED
GAMES

FOR THE
COMMODORE
64[®]

LARRY HATCH

HATCH 25 ADVANCED GAMES FOR THE COMMODORE 64®



Reston

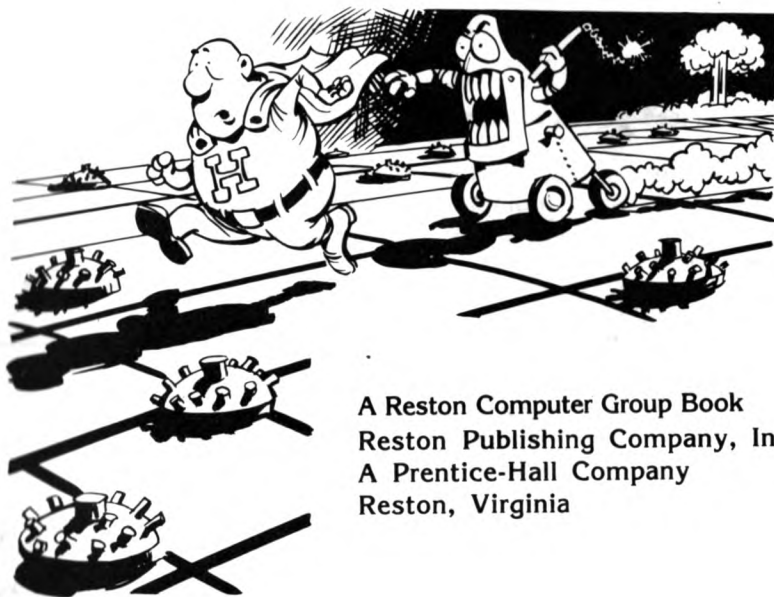


**25 Advanced Games
for the Commodore 64®**



25 ADVANCED GAMES FOR THE COMMODORE 64®

Larry Hatch



A Reston Computer Group Book
Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

The cartoons at the beginning of each program were created by Bruce Bolinger, Freehold Studios, Nicktown, Pennsylvania

Opinions expressed by the author are his own and are not necessarily those of the publisher.

Commodore 64® is a registered trademark of Commodore Business Machines.

Commodore BASIC® is a registered tradename of Commodore Business Machines.

PET® and CBM® are registered trademarks of Commodore Business Machines.

Apple® is a registered trademark of Apple Computer, Inc., Sunnyvale, California.

IBM® Personal Computer is a registered trademark of IBM Corporation.

Library of Congress Cataloging in Publication Data

Hatch, Larry.

25 advanced games for the Commodore 64.

1. Computer games. 2. Commodore 64 (Computer)—Programming. 3. Basic (Computer program language)

I. Title. II. Title: Twenty-five advanced games for the Commodore 64.

GV1469.2.H368 1984 794.8'2 84-6854

ISBN 0-8359-7893-1

Copyright 1984 by Reston Publishing Company, Inc.

A Prentice-Hall Company

Reston, Virginia 22090

All rights reserved. No part of this book may be reproduced in any way, or by any means, without permission in writing from the publisher.

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

TO SUE CARLSON

*for her great patience
while I was pounding the keyboard*



CONTENTS

Preface	ix
System Notes	1
Interface Notes	7
Listing Conventions	9
Programs	
1 Raging Robots, 15	
2 Crab Races, 24	
3 Flatcat, 32	
4 Tic Tec Tac Toc Toe, 42	
5 Crosstracks, 51	
6 Crypto, 60	
7 Keno, 66	
8 Saucers, 77	
9 Blackjack, 84	
10 Magic Cube, 92	
11 Magic Circles, 100	

- 12 3-D Tic Tac Toe, 107
- 13 Gypsy Moth Invasion, 116
- 14 Polyform, 126
- 15 Reds, 134
- 16 Quantum Billiards, 142
- 17 The Thing, 151
- 18 Pocket Puzzle, 160
- 19 Fill the Curve, 167
- 20 BANDIT, 174
- 21 Entrigma, 182
- 22 Decahex, 189
- 23 Passline, 194
- 24 Zinger, 202
- 25 Torpedo, 210

Appendix: PEEK and POKE memory locations	219
Glossary	225

PREFACE

The purpose of this book is to present interesting, well-written BASIC programs which make full use of the many special features of the Commodore 64® (C-64) personal computer. These features include character, sprite and full-screen dot resolution graphics, and a full-fledged sound synthesizer chip, to name a few. At this writing, the C-64 has fallen under \$150.00 in price and represents an undeniable bargain.

These programs are fully documented with line-by-line explanations, variable lists, screen displays, and other helpful information. The programs are densely written to get the most computing out of the shortest LISTs. Most lists are 80 lines or less in length and were run directly from the author's Commodore 64 to the printer, after every effort was made to build in quality and to eliminate bugs.

The games range from strategy to cryptography, from mathematics to torpedoes, and from word recognition to gambling systems. While some games may be mathematically interesting, no profound knowledge of mathematics is needed to understand and enjoy them.

If you have a Commodore 64, this book belongs on your shelf. Maybe you should pick up a second copy for your briefcase.

L.H.
Menlo Park, California
January 1984



**25 Advanced Games
for the Commodore 64®**



SYSTEM NOTES

The most complete information available for the serious programmer remains, at this writing, the Commodore 64 *Programmer's Reference Manual*. Published by Commodore and Howard Sams & Co. Inc., it sells for less than \$20 and seems indispensable.

The memory map of a computer is a list of numbers representing memory locations, along with the functions assigned to those addresses. The memory map of the C-64 is unlike any previous Commodore machine. To start with, more than one device or function can occupy the same group of addresses! Theoretically, you could program all 65,532 bytes to be "RAM" (read-write memory). A language other than BASIC could be loaded where BASIC usually resides and take over the entire system! The 6510 microprocessor chip is a direct descendant of the 6502 device found in earlier Commodore machines. It assigns to the first two locations in its 65,532-byte range (bytes #0 and #1) the task of deciding which devices will occupy and be "attached" to large blocks of addresses in the memory map. Ordinarily, location zero has the number 47 stored, and location #1 has 55 stored. This ensures that the C-64 will wake up speaking to us in BASIC as usual. By playing with these memory contents, we can lose control of the machine, especially location #1.

Memory locations up to 32767 have RAM (random access memory) only. This is undisputed. Locations between 32768 and 40959 can be taken over by a plug-in game cartridge and take over the rest of the system from there. Locations 40960 to 49151 can also be taken over by cartridge programs. If no cartridges are used (and we won't), then your user memory extends to 40959. The BASIC language, in non-erasable Read Only Memory (ROM) will extend upward from location 40960 to 49151.

Next, there are 4096 bytes of (usually unused) RAM from 49152 to 53247. From there upward, BASIC takes over again with 4096 bytes assigned to either Input/Output devices (I/O) or to the character generating ROM. Thus, there are THREE different families of computer guts occupying this string of addresses from 53248 to 57343! Don't feel discouraged. You don't need to know all of this yet.

From location 57344 to the end of memory at 65531 there is another large (8K) block of BASIC called the "Kernal." That is Commodore's spelling, not mine. Loosely speaking, the lowest 8K of BASIC (32768 to 40959) appears to be the "Interpreter." This part of BASIC tries to make sense out of our basic programs and see that they are executed in proper order. The interpreter does this by calling machine language routines up in the "Kernal" which do the actual work. The I/O section enables the machine to talk to the outside world through a screen, keyboard, tape, disk, printer, and anything else considered as a "peripheral."

When you turn on your machine, the BASIC interpreter puts every-

thing in order and puts itself in charge of the system. Memory space up to location 1023 is given to the processor, and to BASIC, as scratchpad memory; keeping track of line numbers, the real time clock and all sorts of small but important things. The screen character space is given 1000 locations from 1024 up to 2024.

The corresponding color memory is the 1000 memory locations from 55296 up to 56295. Both corresponding locations need a number stored to display a character. PRINT statements take care of this automatically. User memory starts after the screen character commencing from 2048 upward to 40960. This, less one byte, is the "38911 BYTES FREE" that the C-64 displays when you power it up. If you have the C-64 LINK interface, this "free" memory drops to 30719 bytes, with the interface behaving like a cartridge. It plugs into the cartridge slot.

While written in standard Commodore BASIC, these programs are full of POKES and PEEKs. The BASIC POKE command tells the machine to unconditionally store a number in a particular memory location. Care must be taken with these POKES. Strictly speaking, PEEK(m) is not a command but a function. N = PEEK(m) is a command that sets the value of the variable N equal to the contents of memory location m. These PEEKs and POKES will activate the sound and video chips for all sorts of effects. Other PEEKs and POKES will trick the machine into performing special stunts that BASIC doesn't usually do.

Let's make the screen more readable. If you just turned on your machine, the whole screen and character set is in barely distinguishable shades of blue. Type in

```
POKE 53281,0
```

(and hit the [RETURN] key). *Voilà*, the screen is black. Now type in

```
POKE 53280,9
```

[RETURN] and the screen border is some excuse for brown. This will be a lot easier on the eyes in the long run. Next, press the control key [CTRL] and hold it down while you press the [WHT] key (the control-shifted [2] key). From now on, everything you type will be in white. By adjusting the color controls on your television or monitor, you can get maximum readability for entering the programs that follow.

Before we continue, a few of the other most common PEEKs and POKES should be explained. In most, if not all of these programs, the constant QA is assigned the value 198 early on. This is the memory location of the keystroke counter down in scratchpad RAM. If we POKE QA,0 and

WAIT QA,1 the machine will wait for the next keystroke before continuing with a program. This eliminates the INPUT statement for one-keystroke action from a game player. The need to press [RETURN] every time is eliminated. Usually, a GET Z\$ statement, or the like, follows to see which key was pressed.

The constant QB is assigned the value 214. Memory location 214 determines which of 25 lines on the screen is to be PRINTed to next. This trick requires a dummy PRINT statement. Type in

```
QB=214:POKEQB,20:PRINT:PRINT"UCUMBERS".
```

When you hit RETURN the cucumbers should appear near the bottom of the screen regardless of where you were PRINTing before.

The constant QC is sometimes used, too. QC is set to 197 or 203, either will do. These two locations, which are constantly updated, store a number corresponding to which key is held down at any moment in time. If no key is being pressed when you PEEK at this location QC, the value 64 will be found stored there.

Other common constants are SC = 1024 which is the beginning of screen character memory and CS = 55296 which is the corresponding screen color memory location. If we POKE a character onto the screen at SC+N, where N is between zero and 999, it will probably not be visible until we also POKE in a color (0 to 15) into location CS+N.

Other familiar constants (like AD,SR,WF which will be explained as we encounter them) are assigned to control sound. There is one curious effect that may work to our benefit. If, for instance, you mistakenly set QD = 198 instead of QA, and later go POKEing numbers into QA, you will be POKEing into location #zero since no value has been assigned to the constant QA. Usually (though not always), this will turn on the drive motor of your cassette unit if it is plugged in. The tape won't turn unless the PLAY key is depressed, but you can hear the motor hum if you listen closely. This is an almost sure sign that there is a mistake in typing in a program. The same thing will happen if you assign QA = 198 correctly and later start POKEing into QF by mistake.

All of these constant names were chosen arbitrarily by me for the sake of consistency. In earlier publications, I used the same names QA, QB, etc., but they were assigned different values to match the machine at hand.

We will POKE numbers into a lot of memory locations, and sometimes a POKE from one game will interfere with another game loaded later. For example, memory location 54278 controls Sustain/Release for sound voice #1. Torpedo and other games will POKE numbers into location 54278 for special effects. Games like Raging Robots do not use Sustain/Release,

and if you load them later, the sound may be affected. A PEEK into sound locations will not reveal their true contents and worse, the STOP-RESET keys will not reset them to zero.

It would be best to add a line to any programs that display surprise sprites or weird sustained sounds:

```
105 FOR ZN=54272 TO 54296:POKE ZN,0:NEXT ZN:POKE 53269,0
```

This will turn off unwanted sound effects and sprites. The correct effects will be turned on by the program lines after line 110.



INTERFACE NOTES

If you choose to connect your C-64 to your color television, then the simple directions in the *Commodore 64 User's Guide* will do. This volume comes with the machine.

In order to get better color and sound, I bought an NEC model C12-202A color monitor made by Nippon Electric Co. This eliminates the difficulties of a television receiving and decoding a radio-frequency channel. McGaffey Engineering of San Jose designed an interface cable to connect it up.

This interface consists of a 5-pin DIN connector that plugs into the rear of the C-64 at the "audiovisual" jack instead of the TV connector to its left. Pins 2, 3 and 4 are used only. Pin 2 is the signal ground, and it connects to the sheaths of two coaxial cables. Pin 3 is "audio output." It connects to the center conductor of the "audio" cable. Pin 4 is the "composite video" output. It connects to the center conductor of the other cable for video. You will want to label the two cables. At the ends of both cables are standard RCA-type (phonograph) plugs that go directly into the monitor's audio and video inputs.

In addition to the monitor, my system has the usual "Datasette" tape recorder and a special interface "dongle" called the C-64 Link. This interface, made by Richvale Telecommunications of Ontario, Canada, enables the C-64 to communicate through the IEEE-488 bus, which connects to all my other Commodore equipment of earlier vintage. This includes a 2022 tractor printer, a model 8050 one-megabyte dual floppy disk system, and a CBM 4032 computer that is word-processing these words as I write them.

The disk system stores both C-64 and PET/CBM programs as well as the word processor programs and files. The CBM (after some strategic POKES to memory) can load, debug, renumber, and otherwise edit the C-64 programs, even though it could never run them.

Both computers run virtually the same Commodore BASIC®. (The C-64 Link device even adds CBM BASIC 4.0 disk commands to the C-64!) Things go wrong when PEEK and POKE commands from one type of machine are used in another, for reasons that will become obvious.

LISTING CONVENTIONS

To prevent having to drastically reduce the size of the lists to appear in this book, line lengths were restricted to 52 characters or less each. Most unnecessary spaces were deleted to reduce memory requirements and gain running speed. The lines are densely packed for the same reason, with as many as half a dozen commands per line number. This, along with the relatively short program lengths, allows all of the programs to run well under 8K of memory.

The **LISTs** are shown exactly as they were printed from the **CBM 2022** printer to prevent errors in printing. This means that cursor, reverse field, and color control characters are shown as the same graphics characters that will appear on your screen when you type in the programs, with exceptions as noted. Unfortunately, these aren't very readable and do not indicate their true functions, so lists of the control characters follow.

All cursor control and color symbols will be **LISTed** in reverse field. They will be enclosed in quotation marks, perhaps along with some text or prompt messages. The **HOME** function is shown as a reverse-field capital letter **S** for instance. The computer will place the cursor "home" at the upper lefthand corner of the screen whenever it finds this character inside of quotes. The reverse-field **Q** is the **CLEAR-SCREEN** function. This entirely clears the screen and takes the cursor home.

The cursor left, right, up, and down keys are at the lower right corner of the keyboard. The **CLR/HOME** key is at the upper right. These keys are used by themselves or with the simultaneous **SHIFT** key. The color and reverse keys, on the other hand, are all entered by simultaneously pressing the **CTRL** (control) key. These keys are actually the number keys across the top of your keyboard, but they are "shifted" with the control key.

My printer cannot reproduce the British pound sterling sign (a script capital letter **L** with a slash through the middle), which tells your computer to print in green. In my lists, instead, you will see a reverse field backslash () that looks like a black box with a downward sloping white line through it. Nevertheless, press **CTRL 3** for green, and the **C-64** will do the rest. Another character that is difficult to make out is the symbol for purple. It will show up as a box that is grey on the left and black on the right. Finally, the sign for yellow is a reverse field Greek letter pi. This symbol is difficult to recognize on the CRT screen.

You will often encounter **PRINT** statements that are full of spaces. These are used to blank out and clean up portions of the screen quickly by covering old screen prompts and messages. Count the spaces carefully so that junk isn't left on the screen. Sometimes, I put numbers into these spaces, with wavy lines (tildes) on either side. This is to tell you how many spaces are required between quotation marks. Just type in the spaces, and don't forget the second quotation mark!

I cannot overemphasize the need to frequently SAVE any program on tape or disk as it is being typed in. Never mind if you haven't finished the LIST. Nothing is more frustrating than losing 50 lines of tedious list entries because of a power outage or an unfortunate keystroke (like the RUN key for instance)!

Never run a program with a lot of POKE statements without SAVEing it first! One of them is likely to be mis-entered. The C-64 has one saving grace. Pressing the STOP and RESTORE keys simultaneously will *usually* bring the machine back from never-never land. Prior to the Vic-20, this key did not exist on Commodore machines.

If you find this book helpful and know someone with any 40-column Commodore PET® or CBM® machine, may I recommend my book *25 Advanced Games for the PET/CBM?* The same general format and many of the same programs appear, especially adapted to those machines. It might be prudent to buy two copies; one could get stolen or something.

COLOR CONTROL CHARACTERS

1	"█"	= PRINT IN BLACK	[CTRL 1]	
2	"▒"	= PRINT IN WHITE	[CTRL 2]	
3	"␣"	←(POUND STERLING SIGN)	RED	[CTRL 3]
4	"▒"	= PRINT IN CYAN	[CTRL 4]	
5	"▒"	= PRINT IN PURPLE	[CTRL 5]	
6	"▒"	= PRINT IN GREEN	[CTRL 6]	
7	"▒"	= PRINT IN BLUE	[CTRL 7]	
8	"▒"	= PRINT IN YELLOW	[CTRL 8]	

CURSOR CONTROL CHARACTERS

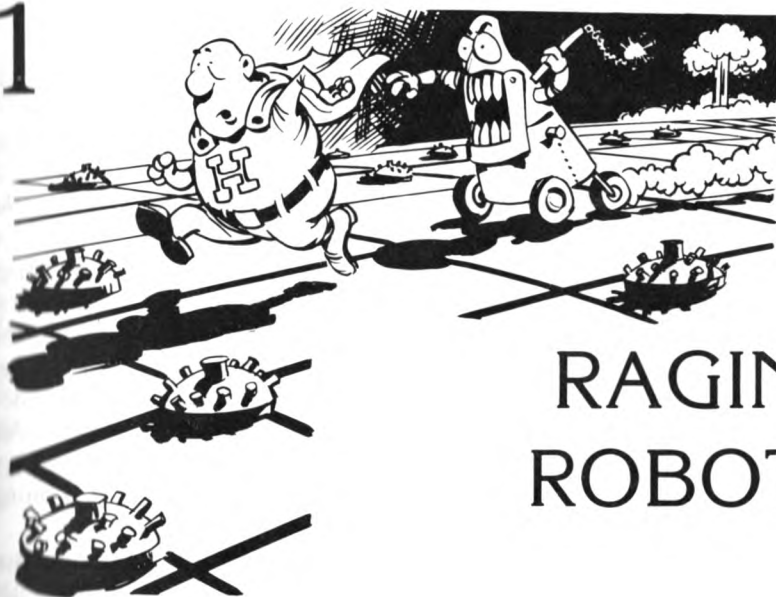
1	"␣"	= CURSOR DOWN KEY UNSHIFTED.
2	"␣"	= CURSOR UP, SAME KEY SHIFTED.
3	"␣"	= CURSOR RIGHT KEY UNSHIFTED.
4	"␣"	= CURSOR LEFT, SAME KEY SHIFTED.
5	"␣"	= CURSOR HOME KEY UNSHIFTED.
6	"␣"	= CURSOR HOME AND CLEAR SCREEN, SAME KEY SHIFTED.
7	"␣"	= PRINT IN REVERSE FIELD; [CTRL] [RVS ON]
8	"␣"	= TURN REVERSE FIELD OFF; [CTRL] [RVS OFF]



PROGRAMS



1



RAGING ROBOTS

This program for the Commodore PET first appeared in *Recreational Computing Magazine* in 1980. Since then, I have written versions for everything from the EXIDY™ Sorcerer to a piece of wafer test equipment called the Lomac 80 at National Semiconductor Corporation in Silicon Valley. The Lomac version was clearly the worst, but even it became popular. In spite of its relative simplicity, Raging Robots is my most popular program. It is amazing to watch Ph.D. candidates playing dozens of games in a row.

Raging Robots consists of 30 robots that appear as crossed white blocks. The player is represented by a "hero" shown on the screen as a solid white ball. The player gets the first move so that he can head toward some advantageous position away from the robots. The robots will each get one move for every move the player makes, and all 30 will advance before the player gets another move. There are 80 landmines scattered about the screen, and these are the player's only protection. The robots will blindly collide with them and be eliminated if they are in the way.

The sole object of the game is survival against the odds, and the player wins when the last robot has been lured into a landmine. Assuming a robot is on a different row and column than the hero, his move will consist of two steps—horizontal and then vertical, one space each. The result is a diagonal move toward the hero. Each game will be different since there is

EXIDY is a trademark of Exidy Incorporated and is copyright 1981, Exidy Incorporated.

an astronomical number of initial positions for all the landmines and robots, and this must be multiplied by another large number of strategies.

Usually, if the hero gets sandwiched between two robots with no landmine protection nearby, the game is lost. Sometimes the robots can be tricked into jumping onto each other, thereby destroying both! Which of two attacking robots moves first determines the success of this strategy. A very simple game, therefore, can become an exercise in short- and long-term strategies. If that sounds a bit dry, I included plenty of sound effects.

The number of robots can be adjusted up or down by changing the end of line 240 from NR=30 to some other number. For an easier game, try 20 robots. Forty robots is the upper limit for any reasonably finite chance of winning, and this should be reserved for the most expert players. For now, I recommend 30 robots as programmed. If you don't win at first, try to take the most robots with you when the hero is lost at the end. A running tally of remaining robots is displayed near the top of the screen.

The sound synthesizer chip is an immense improvement to Commodore's line of computers. Here, we put it to good use. First, the landmines and robots are POKEd to the screen. As we do so, we just turn the volume on and off by POKEing values into memory location VL (= 54296) alternated by POKEing in zeroes. This simple trick gives us little clicks as these characters appear. Later on, as the robots move one by one toward the hero, each motion is accompanied by a clanky mechanical sound that varies with each advance. This is done by POKEing in different pitches into voice one and voice three using "ring modulation."

For ring modulation, we select a waveform by POKEing 21 into the waveform location of voice one (WF = 54276). Twenty-one corresponds to the triangle waveform (16) plus the ring modulator bit, which adds four plus one to activate sound. The ring modulator rings two different pitches against each other for an interesting metallic sound. Again, I recommend the Commodore 64 *Programmer's Reference Manual* for a complete technical description of the SID sound chip.

Another sound effect is the explosion of a landmine. Here I had to experiment until it became realistic. The examples shown in the *Programmer's Reference Manual* sound like anemic cap pistols or champagne corks popping. The trick was to turn bit zero on and off in the waveform byte (WF) by alternately POKEing in values of 129 and 128 separated by a small delay loop. The same crunching explosion sound reappears in the last program in this volume.

While the sound subroutines are at the end of the program, along with other routines, the main loop of the program is between lines 240 and 760. This is broken into two main parts; the player's move and the 30 robots converging for the kill. If the hero loses, a cross is erected in the lines following line 800 to show due respect. After displaying scores, etc., we clear

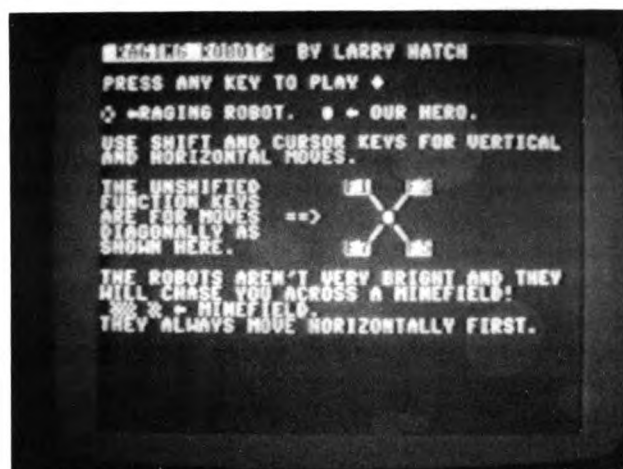
the rather tangled loops and variables using the CLR statement at line 900. This even UN-DIMensions the arrays, so we can start all over with a clean slate starting at line 240. Since all variables and constants are cleared, we POKE the hero and robot scores into locations 894 and 895 where BASIC can't find them.

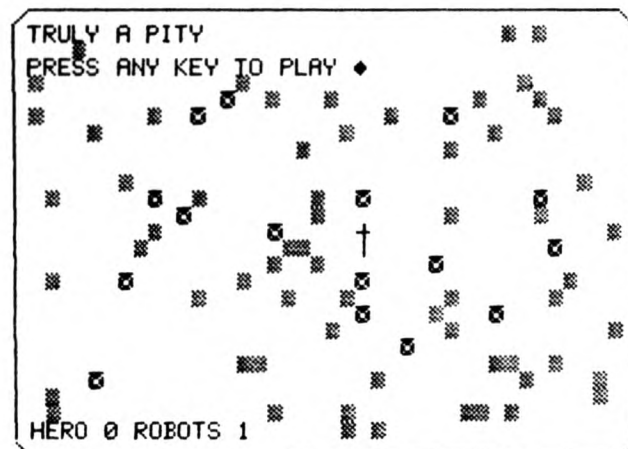
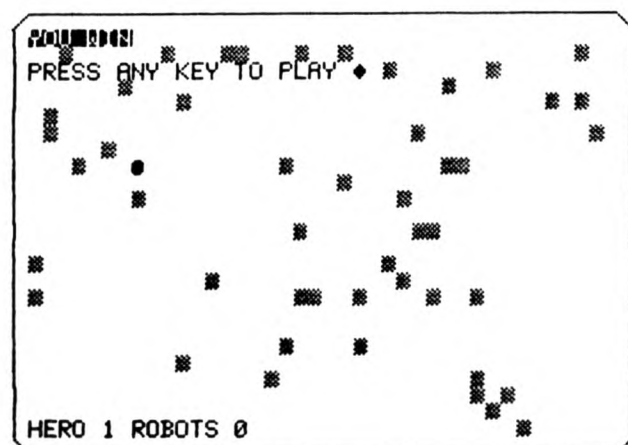
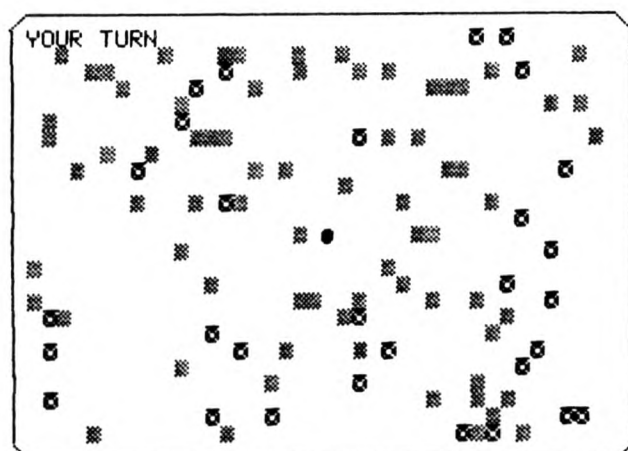
A full line-by-line program analysis follows the LIST, and a list of variables is provided for those who want to study the lines in detail. At the end of this book is a list of PEEK and POKE memory locations to help decipher obscure-looking lines. The glossary may help as well.

If you do start winning the "regulation" game of 30 robots, it is considered daring to try to end the game near the center of the screen. Players start ranking when they win half of the games.

```

████████████████████ BY LARRY HATCH
PRESS ANY KEY TO PLAY ♦
◊ ← RAGING ROBOT. ● ← OUR HERO.
USE SHIFT AND CURSOR KEYS FOR VERTICAL
AND HORIZONTAL MOVES.
THE UNSHIFTED
FUNCTION KEYS
ARE FOR MOVES
DIAGONALLY AS
SHOWN HERE.
    ==>
    (diagonal movement diagram)
THE ROBOTS AREN'T VERY BRIGHT AND THEY
WILL CHASE YOU ACROSS A MINEFIELD!
███ ← MINEFIELD
THEY ALWAYS MOVE HORIZONTALLY FIRST.
  
```







LIST #1: RAGING ROBOTS

```

1 REM*****
2 REM* RAGING ROBOTS    COMMODORE 64 *
3 REM* BY LARRY HATCH   V10.28 *
4 REM* MENLO PARK CALIF 94025 1983 *
5 REM*****
100 POKE 894,0:POKE895,0:POKE53280,9:POKE53281,0
110 CLR:PRINT"RAGING ROBOTS BY LARRY HATCH"
120 PRINT"RAGING ROBOT.  ●  ← OUR HERO."
130 PRINT"USE SHIFT AND CURSOR KEYS FOR VERTICAL"
140 PRINT"AND HORIZONTAL MOVES." :POKE53269,0
150 PRINT"THE UNSHIFTED      F1  F3"
160 PRINT"FUNCTION KEYS      \  /"
170 PRINT"ARE FOR MOVES  ==>  ●  "
180 PRINT"DIAGONALLY AS      /  \"
190 PRINT"SHOWN HERE.        F7  F5"
200 PRINT"THE ROBOTS AREN'T VERY BRIGHT AND THEY"
210 PRINT"WILL CHASE YOU ACROSS A MINEFIELD!"
220 PRINT"  X  X  ← MINEFIELD."
230 PRINT"THEY ALWAYS MOVE HORIZONTALLY FIRST."
240 PRINT"PRESS ANY KEY TO PLAY ♦":NR=30
250 VL=54296:AD=54277:P1=54273:P3=54287:WF=54276
260 QA=198:QB=214:SC=1024:CS=55296:POKEQA,0:ZF$=""
270 RT=TI:WAIT QA,1:GETZF$:POKEQA,0:RQ=TI-RT
280 RR=RQ*NRND(1):DIM R(NR),P(NR):R=NR
290 PRINT"J":FORI=1TO80:POKEVL,5:POKEVL,0
300 MF=INT(988*NRND(RR))+12:POKECS+MF,102
310 POKECS+MF,5:NEXTI:FORI=1TONR:POKEVL,10
320 R(I)=INT(988*NRND(RR))+12:IF R(I)=500THEN320
330 POKE SC+R(I),214:POKECS+R(I),4:P(I)=1

```

```

340 POKEVL,0:NEXTI:YU=500:POKESC+YU,81
350 POKECS+YU,1:POKEQA,0:YS=PEEK(895)
360 IFR=0THENPRINT"YOU WIN":YS=YS+1:GOTO840
370 PRINT"YOUR TURN ":POKEQA,0:WAITQA,1
380 GETD$:YS=PEEK(895):IFD$=""THEN380
390 DA=ASC(D$):IFDA=145THENDP=-40:GOTO480
400 IFDA=17 THENDP=40:GOTO480:REM DOWN
410 IFDA=157THENDP=-1:GOTO480:REM LEFT
420 IFDA=29 THENDP=1:GOTO480:REM RIGHT
430 IFDA=133THENDP=-41:GOTO480:REM UP-LEFT
440 IFDA=134THENDP=-39:GOTO480:REM UP-RIGHT
450 IFDA=135THENDP=41:GOTO480:REM DOWN-RIGHT
460 IFDA=136THENDP=39:GOTO480:REM DOWN-LEFT
470 GOTO360:REM* INVALID KEYSTROKE SO GO BACK
480 YN=YU+DP:IFYNK40ORYN>999 THEN YN=500
490 IFPEEK(SC+YN)<>32 THENLC=YU:GOTO790
500 POKESC+YN,81:POKECS+YN,1:POKESC+YU,32
510 YU=YN:POKEQA,0:TP=10+10*D:GOSUB900
520 PRINT" ":FORI=1TONR
530 IFPEEK(SC+R(I))=32THENP(I)=0:GOTO730
540 IFP(I)=0 GOTO730
550 X1=R(I)-40*INT(R(I)/40)
560 X2=YU-40*INT(YU/40):X=X2-X1
570 Y1=INT((R(I)-1)/40):Y2=INT((YU-1)/40):Y=Y2-Y1
580 J1=R(I)+SGN(X):D1=PEEK(SC+J1):POKESC+R(I),32
590 POKE SC+J1,214:POKECS+J1,4:R(I)=J1
600 J2=R(I)+40*SGN(Y):TP=10+I*2:GOSUB900
610 D2=PEEK(SC+J2):IFX=0 GOTO660
620 IF D1=81 THEN P(I)=0:LC=SC+YU:GOTO 790
630 IF D1=102THEN LC=SC+J1:P(I)=0:R=R-1
640 IF D1=214THEN LC=SC+J1:P(I)=0:R=R-2
650 IFD1=102ORD1=214THENGOSUB770:GOTO730
660 IFP(I)=0ORY=0 GOTO730
670 POKE SC+J2,214:POKECS+J2,4:POKESC+R(I),32
680 R(I)=J2:TP=30+I*2:GOSUB900
690 IFD2=81 THEN P(I)=0:LC=SC+YU:GOTO790
700 IF D2=102THEN LC=SC+J2:P(I)=0:R=R-1
710 IF D2=214THEN LC=SC+J2:P(I)=0:R=R-2
720 IFD2=102ORD2=214THENGOSUB770
730 NEXTI:RC=0:FOR RT=1TO NR
740 IFPEEK(SC+R(RT))<>214THENP(RT)=0
750 IFP(RT)<>0THENRC=RC+1
760 NEXTRT:R=RC:PRINT"R" :GOTO360
770 FORJ=1TO12:POKELC,90:POKELC,32:NEXTJ
780 GOSUB920:PRINT"R" :RETURN
790 J=0:FORJ=1TO15:POKELC,90:POKELC,32
800 NEXTJ:GOSUB920:POKESC+YU,32:POKESC+YN,93
810 POKECS+YN,1:POKECS+YN-40,1
820 POKECS+YN-40,91:RS=PEEK(894)+1
830 PRINT"TRULY A PITY ":POKE894,RS
840 RC=0:POKE895,YS:FOR RT=1TONR
850 IFPEEK(SC+R(RT))<>214THENP(RT)=0
860 IFP(RT)<>0THENRC=RC+1
870 NEXTRT:R=RC:PRINT"R" :POKEQB,23:PRINT

```

```

880 PRINT"HERO"PEEK(895);;"ROBOTS"PEEK(894);
890 CLR:GOTO240:REM*SOUND SUBROUTINES BELOW
900 POKEWF,0:POKEAD,40:POKEVL,15:POKEP1,TP
910 POKEP3,9+PEEK(162)/9:POKEWF,21:RETURN
920 POKEWF,0:POKEAD,7:POKEP1,30:REM*EXPLOSION
930 FORZ=15TO 0STEP-1:POKEVL,Z:POKEWF,129
940 FORQ=0TO2:NEXTQ:POKEWF,128:NEXTZ:RETURN

```

LINE ANALYSIS FOR RAGING ROBOTS

1-5 This is a program header with names, dates, credits, version number, and other data. Consisting entirely of REMarks, it is not executed.

100 Zeroes scores (hero and robot); sets screen field and border colors.

110-230 Clear all variables and print instructions. 110 clears the screen, homes cursor, and prints in white and reverse field. See list conventions for graphic symbols.

240 Cursor goes home and down twice. Prints message. Sets number of robots to 30. This line starts each new game.

250 Sets sound-generation constants.

260 Sets other screen print and keyboard constants.

270-280 Randomize RND function using keyboard delay. Wait for keystroke then DIMension arrays.

290-310 Clear screen and set 80 random landmines.

310-340 Place NR (30) robots randomly on the screen.

340-350 Place hero (white spot) on screen; PEEK his score in.

360 Tests for no robots (a win for the player).

370-380 Wait for player's next move.

390-460 Convert keystroke to hero motion displacement DP.

470 Disallows irrelevant keystrokes.

480 New hero screen position (YN); keeps it on the screen.

490 Detects hero's collision with anything, an immediate loss.

500 Hero POKEd into new screen position. Old position blanked.

510 "New" position becomes "old" position. Noise made.

520 Screen prompt cleared and main robot loop started.

530 Drops absent robots from robot roster array P(I).

540 Skips to NEXT (I+1) robot if the Ith robot is dropped.

550-560 Calculate X-coordinates of hero and Ith robot on screen.

- 570 Calculates hero and robot screen Y-coordinates
- 580 Calculates robot's next position horizontally.
- 590 POKES in new horizontal (X) robot position on screen. Updates robot position array R(I).
- 600 Calculates robot's next vertical (Y) position.
- 610 Watches for collisions; skips ahead if no X motion
- 620 Tests for collision with hero; ends game if so.
- 630-640 Test for collision with landmine or other robot; eliminate colliding robots.
- 650 Jumps to next robot if this one collides.
- 660 Tests if the Ith robot exists and needs a vertical jump.
- 670 POKES in the vertical robot move; blanks old position.
- 680 Updates robot position array; makes another noise.
- 690-710 Test for robot collision with hero, mine, or robot in order.
- 720 Jumps to detonation flash subroutine if collision occurs.
- 730 Ends robot motion; begins robot body count.
- 740-760 Cancel dead robots, count the live ones, and print the count.
- 770-780 Subroutine for detonation flash; call nested explosion sound subroutine.
- 790-830 The hero's demise: crunch! flash!! Print condolences to the screen. Fall into endgame routine.
- 840-890 End of game. Update and display robot count and scores. Clear all variables. Start new game at line 240.
- 900-910 Subroutine; rusty metal robot sound effects.
- 920-940 Subroutine; crunching explosion sound.

VARIABLE LIST FOR RAGING ROBOTS

In order of appearance:

NR = 30; How many robots will be in the game.

VL is the memory location of sound generation volume. POKE VL, (0 to 15)

AD is the memory location of attack/decay for sound envelope.

P1 is the memory location of voice 1 pitch (frequency).

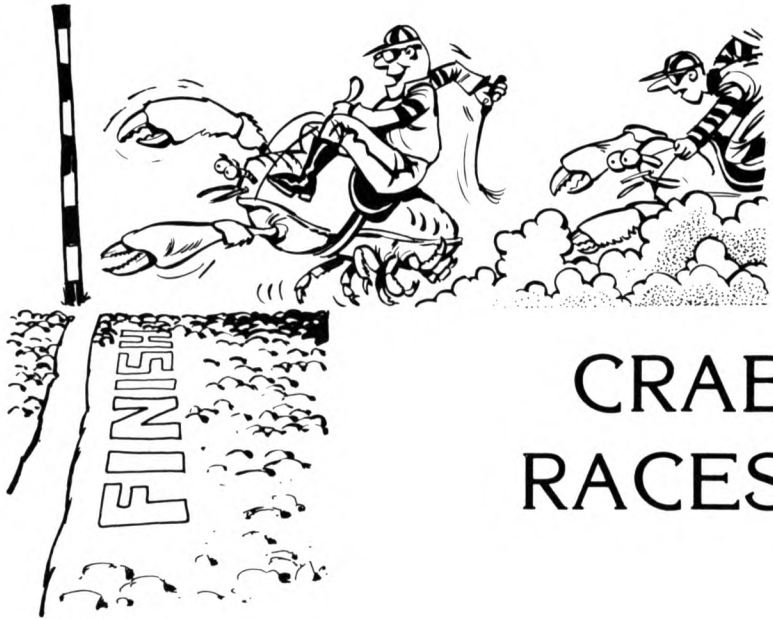
P3 is the memory location of voice 3 pitch (frequency).

WF is the memory location of voice 1 waveform.

QA = 198; Keystroke counter for one-key entries from player.
 QB = 214; Line register for PRINT statements in scratchpad memory.
 SC = 1024; Beginning of screen character memory.
 CS = 55296; Beginning of screen color memory.
 TI is the internal jiffy (1/60 sec.) counter for system realtime clock.
 RR, RT, and RQ count jiffies before a keystroke to randomize the RND function better.
 R(n) = (12-999); The screen position of the nth robot.
 P(n) = (0 or 1); Whether the nth robot is dead (0) or alive (1).
 R = 0 to 30; The robot counter; how many left alive, etc.
 MF = (12-999); Random screen positions for minefields at the start.
 I, J and Z; General-purpose index counters in FOR-NEXT loops.
 YU = (40-999); The hero's present screen position.
 YS; The player's present score. PEEKed and POKEd into location 895.
 D\$; Keystroke of the player's move; a cursor or function key.
 DA; ASCII code for D\$, used in IF-THEN statements to determine DP.
 DP; Displacement number to add to YU to move the hero one square away.
 YN; Player's new position. $YN = YU + DP = (12 \text{ to } 999)$.
 LC; Screen position (12-999) of a collision for visual effects.
 TP; A pitch variable for sound subroutine at line 900.
 X1 and X2; X-coordinates of Ith robot and hero for comparison.
 Y1 and Y2; Y-coordinates of Ith robot and hero for comparison.
 X and Y; The horizontal and vertical distances between robot and hero.
 J1 and J2; New screen positions of robot after horizontal and vertical jumps.
 D1 and D2; Old contents of positions J1 and J2 to test for collisions.
 RC; Another robot counter to count up the live ones.
 RT; Used at the end to index the robot count from 1 to NR (=30).
 RS; The robot's score in games. POKEd into location 894.

Other variables may be used locally as needed and have no set purpose.

2



CRAB RACES

In the desert regions of the USA before all the modern amusements were developed, people would invent their own games using whatever resources were handy. There were lots of lizards scampering about, so lizard races became a natural pastime. That became the theme of the predecessor to this game, Lizard Races. With the sprite graphics and other features of the Commodore 64, it seemed like a perfect idea to rewrite essentially the same program. This idea ran into a snag. For some reason, none of my lizard sprites looked like lizards. They all looked like bugs or crabs. I checked with a friend raised in San Francisco, and by very good fortune, it turns out you can race crabs, too!

On board a fishing boat, they race in all directions toward the rails. On Fisherman's Wharf, they can be raced right up the sidewalk if there aren't too many people around. (Now that has to be fresh crab!) As is true of lizard racing, the crabs all come to a halt in the middle of a race for reasons only crabs know. That is when the most animated wagering takes place. Not knowing at the start which is the faster crab, they all get about the same odds. Once they stop, some crabs will be ahead, others lagging. This invites all kinds of oddsmaking. One crab guarding the rear might be a "sleeper," that is a thoroughbred crustacean who has not yet begun to trundle!

A bet on a crab like that can get odds of 20 to 1 or better. After these mid-race bets, the crabs are encouraged to continue racing. Opeing a jar of locally made crab louis dressing usually does the trick. This mid-race betting can happen 3 or 4 times before the crabs cross the finish line. By betting second favorites, as they trade places with the leader, you can sometimes come out ahead no matter which crab wins.

One phenomenon of race betting is that the payouts on longshots are never as large as the chances of the animal winning are small. Racetracks and bookmakers typically offer 20 to 1 on animals that don't have a chance in 1000 of winning. In this game, I try to rectify this in two ways. First, the crabs have a perfectly even likelihood of being advanced toward the finish line all during the race. Second, a mathematical algorithm is used to approximate the actual chances of any crab coming in, and this determines the odds paid.

A true evaluation of the probabilities of a given crab, racing with six other crabs, when they are many pixels from the finish line involves factorial numbers so large that your C-64 cannot compute them. The task would tax a mainframe computer. Instead, I devised a formula that approximates these probabilities well enough to be more than fair. I would scribble it here, but there isn't enough space on the margin of this page.

The results of these calculations seem reasonable. Far from the finish line, differences in positions have little effect on the odds. There is still plenty of space, and chance, for a slowpoke to catch up. Near the finish line, small differences in relative position cause great differences in the odds. A crab several lengths behind the leader can offer odds of 1000 to 1 or more, and darn it, he should!

The player can choose from two to seven crabs to race. (I set aside the eighth sprite for a jar of crab louis dressing, but it turned out to look like a lizard.) There are eight laps to the race, each with the opportunity to make bets. It is the most fun when the crabs trade off the leading position. In each lap, a different waveform is chosen for the "crab noises," four different sounds in rotation. The ninth lap decides the race.

The object of the game is to second guess the odds to build and retain a positive player's bankroll. The player starts with a zero balance, and he can easily see if he is ahead or behind by the sign of the numbers. At the end of any race, the player can choose to race a different number of crabs by pressing the "C" key when prompted for the next game. Good luck!

CRAB RACES		
1	✠	132.3
2	✠	119.0
3	✠	111.2
4	✠	133.3
5	✠	144
6	✠	176
7	✠	117.5

CRAB #2 4

CRAB RACES		
1	✠	132.9
2	✠	19.6
3	✠	17.1
4	✠	14.5
5	✠	17.1
6	✠	15.1
7	✠	18.7

CRAB #2 4
4.5 TO 1. WAGER? (1-9) 4

LIST #2: CRAB RACES

```

1 REM*****
2 REM* CRAB RACES      COMMODORE 64 *
3 REM* BY LARRY HATCH   V 10.28 *
4 REM* MENLO PARK, CALIF.      1983 *
5 REM*****
100 CLR:PRINT"␣"TAB(14)"CRAB RACES␣"

```

```

110 PRINT"■":POKE53280,8:POKE53281,7:QA=198
120 QB=214:RESTORE:FORI=0TO127:READD
130 POKEI+832,D:NEXTI:FORI=0TO6:LS(I)=2040+I
140 POKELS(I),13:X(I)=53248+2*I:T(I)=13:NEXTI
150 VL=54296:P1=54273:P3=54287:AD=54277:WF=54276
160 FORI=53248TO53264:POKEI,0:NEXTI
170 PRINT"IN CRAB RACES THE CRABS STOP FROM TIME"
180 PRINT"TO TIME; THAT IS WHEN YOU BET ON THEM.■"
190 PRINT"ALL BETS ARE RECORDED AT THE ODDS SHOWN"
200 PRINT"WHEN THE BETTING TOOK PLACE.":POKEQB,20
210 PRINT:PRINT"HOW MANY CRABS (2-7)? ■■■";
220 POKEQA,0:WAITQA,1:GETL$:PRINTL$:L%=VAL(L$)-1
230 QQ=17:IFL%<10RL%>6THENPOKEQB,20:GOTO210
240 PRINT"■":TW=0:LN=L%+1:P=2*LN-1:POKE53269,P
250 FORI=0TOL%:POKE53249+2*I,67+16*I:D(I)=250
260 POKE53248+2*I,50:POKE53287+I,I:NEXTI:BL=80
270 POKE53290,8:PRINT"■"TAB(14)"■CRAB RACES■"
280 F$="-----"
290 POKEWF,0:POKEVL,15:POKEAD,8:POKEWF,129
300 PRINTF$:FORI=0TOL%:H(I)=50:PRINTI+1;TAB(31)"|"
310 PRINTF$:NEXTI
320 R=INT((1+L%)*RND(1)):H(R)=H(R)+3
330 POKEX(R),H(R):T(R)=27-T(R):POKELS(R),T(R)
340 POKEWF,0:POKEVL,10:POKEAD,40:POKEP1,H(R)/2
350 POKEP3,10+2*R:POKEWF,QQ
360 IF H(R)>=BL THEN GOSUB390:REM* ODDS MAKING
370 IF H(R)>=254THEN 610:REM* END OF RACE
380 GOTO320:REM*↑↑ NEXT LIZARD LEAPS
390 DY=EXP(PI):QQ=QQ+2:IFQQ>23THENQQ=17
400 EX=-SQR(DY):SL=0:FORJ=0TOL%:D(J)=254-H(J)
410 CH(J)=D(J)*EX:SL=SL+CH(J):NEXTJ:PRINT"■■■"
420 FORJ=0TOL%:P(J)=CH(J)/SL:O(J)=1/P(J)
430 O(J)=INT(10*(O(J)+.06))/10
440 IF O(J)>35THEN O(J)=INT(O(J))
450 NEXTJ:FORJ=0TOL%:POKEQB,(J+1)*2
460 O$=STR$(O(J)):L=LEN(O$):O$=RIGHT$(O$,L-1)
470 FORI=LT04:O$=O$+" ":NEXTI:PRINT
480 PRINTTAB(33);O$:NEXTJ:BL=BL+25
490 POKEQB,18:PRINT:PRINTTAB(15)"CRAB #? ■■■";
500 POKEQA,0:WAITQA,1:GETY$:REM* BETTING
510 C=VAL(Y$)-1:IFY$="*"ORY$="R"THEN580
520 PRINTY$ " ":IFC<0 ORC>L%THEN490
530 PRINTO(C)"TO 1. WAGER? (1-9) ■■■":POKEQA,0
540 WAITQA,1:GETW$:PRINTW$ " ":W=VAL(W$)
550 Q=Q-W:C(C)=C(C)+W*O(C):POKEQB,18:PRINT
560 PRINT " ":TW=TW+W
570 PRINT " PRESS THE ^*^ TO RUN ■■■":GOTO490
580 POKEQB,18:PRINT:FORI=1TO3
590 PRINT " — 36 — "
600 POKEWF,0:POKEWF,21:NEXTI:RETURN
610 POKEQB,18:PRINT:PRINT"■WINNER #"R+1
620 POKEWF,0:POKEVL,15:POKEAD,8:POKEWF,129
630 Q=Q+C(R):Q=INT(10*Q+.0001)/10
640 PRINT"WINNINGS="C(R); "BETS="TW; "BANK="Q

```

```

650 FORI=0TOLZ:C(I)=0:NEXTI:PRINT"NEXT RACE? ♦♦";
660 POKEQA,0:WAITQA,1:GETZ$:PRINTZ$
670 IFZ$="C"THENPOKEQB,20:GOTO210
680 GOTO240:REM* NEXT RACE, SAME NUMBER RUNNING
690 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,195,64
700 DATA 1,129,128,3,0,192,3,255,192,63,255,252
710 DATA 255,255,255,63,255,252,3,255,192,3,0,192
720 DATA 6,0,96,12,0,48,0,0,0,0,0,0,0,0,0
730 DATA 0,0,0,0,0,0,0:REM* CRAB #1 ↑
740 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,13,0,196
750 DATA 6,0,96,3,0,192,3,255,192,63,255,252
760 DATA 255,255,255,63,255,252,3,255,192,3,0,192
770 DATA 1,129,128,2,195,64,0,0,0,0,0,0,0,0,0
780 DATA 0,0,0,0,0,0,0:REM* CRAB #2 ↑

```

LINE ANALYSIS FOR CRAB RACES

- 1-5 The usual program header with title, credits, etc.
- 100-120 Clear and set constants, POKE in screen colors, etc.
- 120-130 RESTORE data pointer to first item; READ in sprite image DATA and POKE it into the now idle cassette buffer memory locations.
- 130-140 Point seven sprites #(0 to 6) at image #1 starting at location 832. Assign sprite X-position array to controlling registers.
- 150 Sets sound memory location constants as usual.
- 160 Zeroes out all sprite X,Y coordinates.
- 170-200 Print instructions; POKE in next line to print on into QB.
- 210-230 GET keystroke for how many crabs to race. Set QQ=17 for triangle waveform. Disallow irrelevant keystrokes.
- 240 Clears screen. Turns on the right sprites.
- 250-260 POKE sprites (X,Y) into the starting line; POKE each a different color. Set first betting line at 80 pixels (dots) in.
- 270 POKES sprite 3 a better color. Reprints game title.
- 280 Prefabricates a fence as F\$.
- 290 Shoots starting gun with sound POKES.
- 300-310 Install fences to delimit tracks. Assign position array H(I) to each crab. PRINT finish line(s).
- 320-380 Main crab random advancement loop.
- 320 Selects random crab #R. Increments his position H(R).
- 330 POKES crab R into new position. Toggles his sprite image number (13 <-> 14); points sprite R to his new image.

340-350 Make crab noises.
 360 Tests if crab R is at or beyond the betting line.
 370 Tests if crab R won already.
 380 Goes back for a new random crab.
 390-600 Subroutine; oddsmaking and betting.
 390 Math; rotates crab-noise waveform (17, 19, 21, 23, 17 . . .)
 400-410 Establish relative chances of each crab winning.
 420-430 Set probability/odds against each crab.
 440-480 Round off, refine, and PRINT odds for each crab. Move the new betting limit to the right.
 490-500 Player selects crab to bet on.
 510-520 Disallow bad or invalid keystrokes.
 530 Reprints odds on crab #C. Prompts for a wager \$(1 to 9).
 540-550 GET the wager. Multiply wager by odds. Add to that crab's possible payout.
 560-570 Clear old prompt. PRINT new one. Increment total wagers. Go back for another wager if any.
 580-600 Clear up all obsolete prompts. Make a noise. RETURN from subroutine for the next lap of the race.
 610-630 End-of-race routine; announce winner; fire a gunshot. Increment player's bank by Rth crab's purse if any. Round off the player's bankroll.
 640-650 PRINT results. Clear the crab purses. Prompt for next race.
 660 GETs player's keystroke to continue.
 670-680 Test for "C", the signal to change the number of crabs racing next time. Branch back earlier if so.
 690-END Two 64-byte sprite images as DATA to be READ in. The 64th and 128th bytes are dummy numbers to make the 63 byte images consecutive.

VARIABLE LIST FOR CRAB RACES

In order of appearance:

QA = 198; Memory location of keystroke counter.

QB = 214; Memory location of line-print register.

D; Each number READ from the DATA statements to create sprites.

LS(n) = 2040 + n; Memory location of the pointer to the nth sprite's 64 byte image, somewhere in memory. The actual starting location is 64 times the contents of LS(n).

X(n); Memory location of the nth sprite's X-coordinate on screen.

T(n) = 13 or 14; One of two numbers to POKE into LS(n) to alternate sprite images for the nth crab sprite.

VL = Memory location of sound volume.

P1 = Memory location of pitch for voice #1.

P3 = Memory location of pitch for voice #3.

AD = Memory location of attack/decay control for voice #1.

WF = Memory location of waveform for voice #1 (and special effects).

I and J are general-purpose FOR-NEXT loop counter/indexers.

L\$; A player's keystroke asking for 2 to 7 crabs in the race.

L% = VAL(L\$-1); The actual number minus one.

LN = VAL(L\$); For purposes of setting the value of P.

P; A binary number that turns on the right number of sprites.

D(n); The distance of the nth crab to the finish line in pixels.

BL; The limiting value of the most advanced crab, in pixels, before racing halts and betting takes place. BL is regularly increased.

TAB(14) is NOT a number array but an instruction to TAB to the right.

F\$; Creates a fence to place between tracks on the racecourse.

H(n); The X-coordinate of the nth crab in pixels; how far down the track the nth crab is.

R; A random number between zero and L%. It advances the (R + 1)th crab and indexes his arrays.

DY = EXP(pi) = 23.14069 . . . ; Used in oddsmaking formula.

EX = -SQR(DY) = -4.81047 . . . ; Oddsmaking constant. Interesting.

D(J) is the Jth crab's distance to the finish line in pixels.

CH(J) is a relative value proportional to the Jth crab's chances of winning the race (approximate).

P(J) is the probability (0 to 1) of the Jth crab winning.

O(J) = 1/P(J) are the odds against the Jth crab coming in first.

SL is the sum of the relative chances CH(0 to L%).

O\$; The processed string presentation of the odds for neat columns.

L; The length of O\$ string.

Y\$; A player's keystroke to choose his favorite crab for betting.

C; The value of Y\$, less one, to match numbers from zero to L%.

W\$,W; The player's wager on the crab of his choice.

C(C); The sum of all bets on the Cth crab, each one multiplied by the odds against that crab then prevailing.

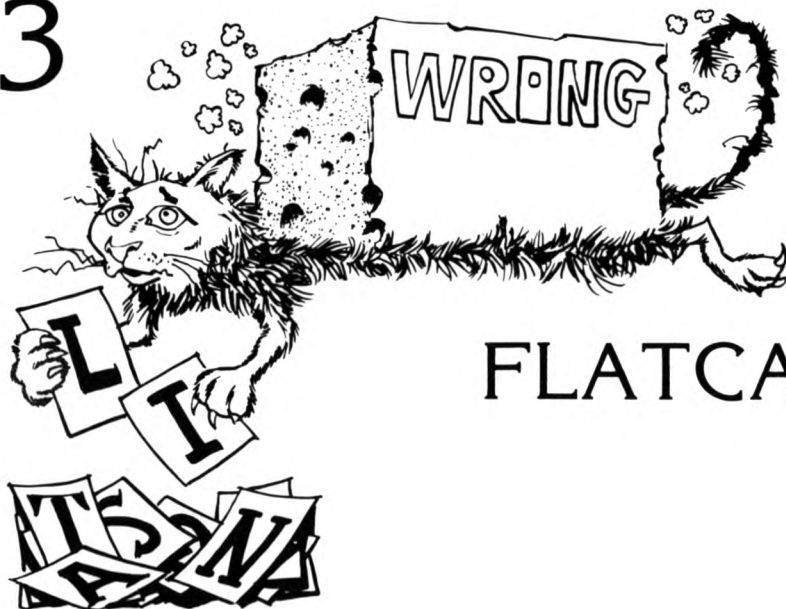
Q; The player's bank; diminished by wagers, added to from winnings.

TW; Total wagers for final statistics after the race.

Z\$; Player's keystroke to start the next race.

Other variables may be used locally.

3



FLATCAT

Flatcat is a vocabulary type word game similar to Hangman but using a different display. Instead of the usual bodily parts of a hanged man appearing in place one by one, we start with a cute little cat. See the first screen display. The player is asked if he wants instructions, and then the game starts.

A colored strip appears at the upper lefthand portion of the screen, with one square representing each letter in the unknown word. The player then guesses a letter in the word. If he guesses a letter that belongs, it appears in correct sequence, leaving the other unknown letters blank.

If he guesses a letter that is wrong, the computer drops a rock on the cat's head. Don't be too upset; it's a small rock. (See the second screen display for this visual effect.)

The player has a number of chances to guess the entire word, which is what he must do to save the cat. The number of guesses depends on the length of the word. Short words are allowed more guesses. They are intrinsically harder than long words! If the entire word cannot be determined, the cat suffers the consequences. See the third screen display.

The rocks are in screen motion with appropriate sound effects generated.

The number of guesses is not limited as long as the guesses are correct. Incorrect guesses are limited to 18 minus the length of the word in letters. For example, the word *broccoli* has eight letters, so a maximum of ten incorrect guesses is permitted. A count of the remaining bad-guess allotment is shown on the screen at all times. Since ten bad guesses are permitted in this example, if all your letters are correct, you gain all ten points plus a bonus point for good measure. Counts are kept of all points scored and all possible points so that a running percentage score can be accumulated from word to word.

This scoring method is rather exacting. It depends upon luck to some degree so that a score of 33% may be considered "good" for the average player. It is possible to save the cat every time and still get very low percentages because the words were discovered at the last minute. While somewhat strict, this scoring method leaves lots of room at the top end of the scale for people with high verbal skills.

If some leeway is needed with inexperienced players, the words chosen can be made longer and more familiar. The author randomly chose everything from easy words to the hopelessly obscure (*zymurgy*?) for the DATA statements at the end of the program. He hopes you will make up your own lists suited to your own needs. The program automatically adjusts for this by counting the items in all of the DATA statements and setting the variable N to account for this all through the program.

A more demanding program might consist of technical terms or a foreign language vocabulary. It might be fun for a group of people to each contribute a half-dozen of their own favorite words and have them typed in by a third party. Remember that the last word in any list must be *zymurgy* for the program to read the word list correctly.

If a player cannot recognize the word at all, he will make one last guess incorrectly. This is the end of the cat. A large granite block drops from above to polish him off. Here the special features of the Commodore 64 stand out again. The sound generator creates a bone crunching squish while a red sprite graphic simulates a puddle of blood emanating from beneath the granite block. A definite improvement over the now classical Hangman game!

While the LIST looks like a very long typing job, remember that one third of it is the DATA statements. These are the words to be used, and you may want to create your own vocabulary. This is fun, and there is no need for the words to be in any particular order, as long as the last one is *zymurgy*.

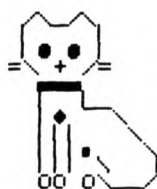
00000000000000000000

6

GUESS A LETTER! ♦

EXACTLY!!! YOU HAVE SAVED THE CAT.

65 POINTS
65 SO FAR
TRY AGAIN OK? P

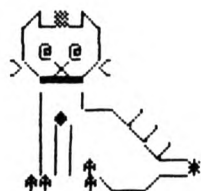


ERSM

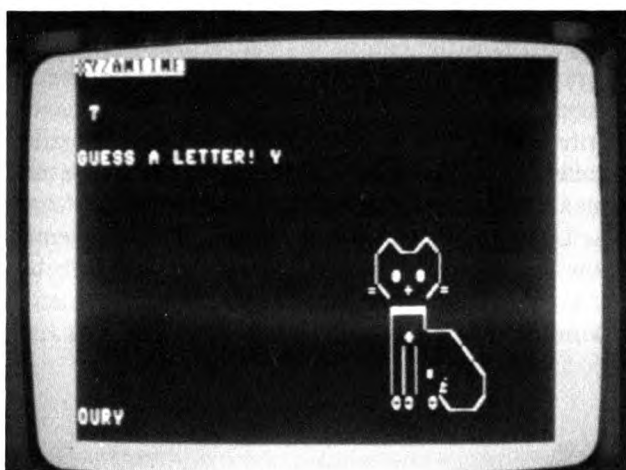
00000000000000000000

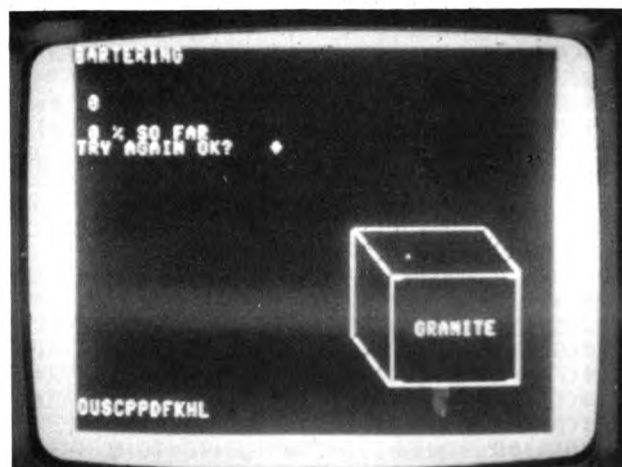
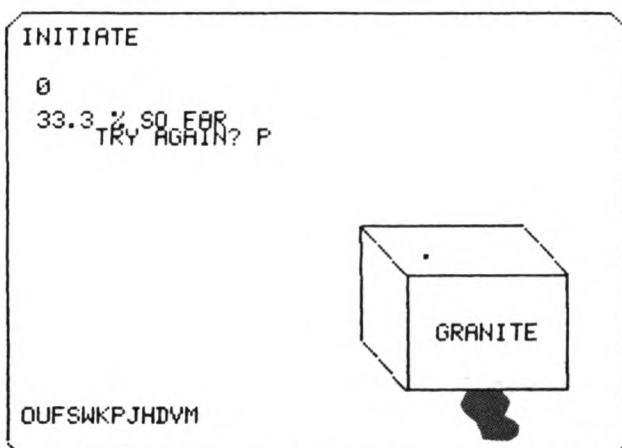
5

GUESS A LETTER! ♦



OUF5WKP





LIST #3: FLATCAT

```

1 REM*****
2 REM* FLATCAT!      COMMODORE 64 *
3 REM* BY LARRY HATCH      C/S *
4 REM* MENLO PARK CA 94025  10.30*
5 REM* ADAPTED FROM HANGMAN  1983 *
6 REM*****
100 CLR:SC=1024:CS=55296:QA=198:QB=214
110 VL=54296:RESTORE:POKE53281,0:DIMC$(25)
120 P1=54273:P3=54287:AD=54277:SR=54278:WF=54276
130 DIMW$(170),CT$(11),CR$(11),FC$(12),L$(25)

```

```

140 PRINT"⌈⌋ FLATCAT ⌈⌋INSTRUCTIONS? ⌈⌋";
150 POKE53280,14:POKE53281,0:POKE53252,0:POKE0A,0
160 WAIT0A,1:GETY$:PRINTY$:IFY$="N"THEN270
170 POKE53269,0:IFY$<"Y"THEN100
180 PRINT"THIS GAME IS MUCH LIKE HANGMAN WHICH"
190 PRINT"MEANS THAT A WORD IS MASKED IN GREY AND"
200 PRINT"YOU GUESS THE LETTERS THAT ARE IN IT."
210 PRINT"IF YOU GUESS CORRECTLY THE LETTER WILL"
220 PRINT"APPEAR IN ITS PROPER PLACES IN THE WORD"
230 PRINT"IF YOU GUESS A LETTER WRONGLY THEN THE"
240 PRINT"COMPUTER DROPS A SMALL ROCK ON THE CAT."
250 PRINT"IF YOU MISS TOO MANY TIMES THE COMPUTER"
260 PRINT"WILL DROP A LARGER PIECE OF STONE."N=0
270 N=N+1:READW$(N):IFW$(N)<"ZYMURGY"THEN270
280 POKE2042,14:FORI=0TO62:READSP:POKE896+I,SP
290 NEXTI:POKE53289,2:POKE53269,4
300 CT$(1)="⌈⌋":PRINT
310 CT$(2)="⌈⌋":REM* "⌈⌋" = YELLOW
320 CT$(3)="⌈⌋"
330 CT$(4)="⌈⌋"
340 CT$(5)="⌈⌋"
350 CT$(6)="⌈⌋"
360 CT$(7)="⌈⌋"
370 CT$(8)="⌈⌋"
380 CT$(9)="⌈⌋"
390 CT$(10)="⌈⌋"
400 CT$(11)="⌈⌋"
410 REM* CAT #2 ↑ PRINT IN WHITE
420 CR$(1)="⌈⌋":REM* "⌈⌋" = GREEN
430 CR$(2)="⌈⌋"
440 CR$(3)="⌈⌋"
450 CR$(4)="⌈⌋"
460 CR$(5)="⌈⌋"
470 CR$(6)="⌈⌋"
480 CR$(7)="⌈⌋"
490 CR$(8)="⌈⌋"
500 CR$(9)="⌈⌋"
510 CR$(10)="⌈⌋"
520 CR$(11)="⌈⌋"
530 REM* THE FLAT CAT. ↑ PRINT IN WHITE
540 FC$(1)="⌈⌋"
550 FC$(2)="⌈⌋"
560 FC$(3)="⌈⌋"
570 FC$(4)="⌈⌋"
580 FC$(5)="⌈⌋"
590 FC$(6)="⌈⌋"
600 FC$(7)="⌈⌋"
610 FC$(8)="⌈⌋"
620 FC$(9)="⌈⌋"
630 FC$(10)="⌈⌋"
640 FC$(11)="⌈⌋"
650 FC$(12)="⌈⌋":REM CRSR UP
660 Z=RD(-TI/1000):PRINT"XPRESS ANY KEY TO PLAY"
670 GETZ$:IFZ$=""THEN670

```

```

680 G=G+1:IFG>=NTHENPRINT"JIM OUT OF WORDS." :END
690 GS=0:R=INT(N*RND(1)+1):IFW$(R)=" "THEN690
700 PRINT"J";X$=W$(R):W$(R)=" ":L=LEN(X$)
710 FORI=1TOL:L%(I)=1:C$(I)=MID$(X$,I,1):PRINT"***";
720 NEXTI:PRINT:GL=18-L:SP=GL+1
730 POKEQB,5:PRINT:PRINT"GUESS A LETTER! ♦♦♦";
740 POKEQB,11:PRINTG$:FORK=1TO11:PRINTTAB(23)CT$(K)
750 NEXTK:POKEQA,0:WAITQA,1:GETG$:IFG$="♦"THEN730
760 PRINTTAB(GS)G$:POKEQB,6:PRINT:PRINTSPC(21)
770 DK=1:A=ASC(G$):IFA<65ORA>90ORG$=" "THEN730
780 POKEWF,0:FORI=1TOL:GA=ASC(G$)
790 IFC$(I)=G$THENPOKESC+I-1,GA+64:L%(I)=0:DK=0
800 NEXTI:D=9:FORI=1TOL:IFL%(I)=1THEND=0
810 NEXTI:IFD=9THEN860:REM* PLAYER WINS
820 IFDK=1 AND GS>GLTHEN990:REM* SQUISH
830 IF DK=0THENGOSUB1070:GOTO730
840 GS=GS+1:SP=SP-1:GOSUB920
850 PRINT"#####"(GL-GS+1)"♦♦ " :GOTO730
860 POKEQB,7:PRINT:REM  ↑ CURSOR LEFT
870 PRINT"EXACTLY!! YOU HAVE SAVED THE CAT."
880 PRINTSP" POINTS":TP=TP+SP:GT=GT+GL
890 PC=INT(1000*TP/GT)/10:PRINTPC"% SO FAR."
900 PRINT"TRY AGAIN OK? ♦♦♦":POKEQA,0:WAITQA,1
910 GETY$:PRINTY$:POKE53252,0:GOTO680
920 SC=1024:POKEWF,0:FORK=1TO12:REM* DROP A ROCK
930 P=SC+27+K*40:POKEP,102:POKEP-40,32
940 POKECS+27+K*40,1:NEXTK:REM* CLONK!
950 POKEP1,SP+55:POKEP3,SP+13:POKEAD,7
960 POKESR,0:POKEVL,15:POKEQB,11:PRINT
970 POKEWF,21:FORK=1TO11:PRINTTAB(23)CR$(K)
980 NEXTK:POKEWF,20:RETURN:REM* BLOCK
990 PRINT"♦";:FORK=1TO11:PRINTTAB(22)" "
1000 PRINTTAB(22)" " :NEXTK:POKE QB,10
1010 PRINT:POKEVL,15:POKEAD,15:POKESR,240
1020 POKEP1,10:POKEWF,129:FORK=1TO12
1030 POKEP1,8*K:PRINTTAB(19),FC$(K):NEXTK
1040 POKE53252,255:POKE53253,226:PRINT"♦"X$
1050 POKEWF,0:GT=GT+GL:PC=INT(1000*TP/GT)/10
1060 POKEQB,4:PRINT:PRINTPC"% SO FAR":GOTO900
1070 POKEVL,10:POKEAD,80:POKESR,0
1080 POKEP1,80-2*SP:POKEWF,17:RETURN
1090 DATA DIRIGIBLE,TURNIP,HORTICULTURE,ASTRONOMY
1100 DATA ALUMINUM,HARPOON,STROBOSCOPE,MYSTERIOUS
1110 DATA HYPOTENUSE,EVOLUTION,PHLOGISTON,VOLCANO
1120 DATA PHILOSOPHY,ABBREVIATION,PHILATELIST
1130 DATA PROMINENT,ALCATRAZ,ALKALINITY,CARROTS
1140 DATA ACCESSORY,ACCORDION,ACETYLENE,ACQUIRE
1150 DATA ACUPUNCTURE,ADAMANT,ADHESIVE,ADMINISTER
1160 DATA ADVANCE,AERONAUTICS,AGRARIAN,ALDERMAN
1170 DATA ALBATROSS,ALLIGATOR,AARDVARK,AMBASSADOR
1180 DATA WHEEL,PEEL,MARIMBA,GONG,CELESTE,UKELELE
1190 DATA AXIOM,AUSPICIOUS,AUSTRALIA,AUTOMATION
1200 DATA BARTERING,BACTERIA,BAGPIPES,BUTTERFLY
1210 DATA BENEVOLENT,BARBARIAN,BOTTLE,BAROMETER

```



```

1220 DATA BYZANTINE, GOTHIC, DORIC, IONIC, ROMAN
1230 DATA CATASTROPHE, CHANDELIER, CZECHOSLOVAKIA
1240 DATA DISCONTINUE, DINOSAUR, DROMEDARY, DYNAMIC
1250 DATA ECCENTRIC, ECHELON, EDUCATION, ELASTICITY
1260 DATA ELECTRICITY, EQUANIMITY, EXCITED, EQUALITY
1270 DATA MUSTARD, RELISH, PICKLES, MAYONNAISE, CATSUP
1280 DATA EAGER, EAGLE, EXCHANGE, EVERGREEN, EYRIE
1290 DATA FASCINATE, FEATHER, FLASHLIGHT, FLUORESCENT
1300 DATA FRAGRANT, FLAGRANT, FLUORESCENT, FLOWER
1310 DATA GIGANTIC, GROCER, GOURMANDIZE, GUACAMOLE
1320 DATA GROTESQUE, GLOVES, GRASP, GREEN, GUITAR
1330 DATA HABITATION, HARMONICA, HAZARDOUS, HEARING
1340 DATA HEXAGON, HICKORY, HEIROGLYPHICS, HIDEBOUND
1350 DATA HARE, HUCKLEBERRY, HUMOROUS, HYDROMETER
1360 DATA OXYGEN, HYDROGEN, NITROGEN, TELLURIUM
1370 DATA DUODENUM, CRANIUM, URANIUM, PALLADIUM
1380 DATA ISOMER, ISOBAR, ISOTOPE, ISOPROPYL, INGOT
1390 DATA IMAGINATION, IMMEDIATE, INTEREST, ISSUE
1400 DATA IMPORTANT, INAUGURATION, INITIATE, INK
1410 DATA INSULATION, IRRIGATION, WOOL, WALL, WATER
1420 DATA JARGON, BARGAIN, JUXTAPOSE, JUBILATION
1430 DATA KING, KANGAROO, KNIGHT, KOALA, KEEPER
1440 DATA LABORATORY, LUNCH, LIQUID, LEMON, LAVA
1450 DATA ARBITRARY, TUNNEL, LAMINATED, WHEEL
1460 DATA WARTHOG, XYLOPHONE, YODEL, ZYMURGY
1470 REM# PUDDLE SPRITE (63 BYTES)
1480 DATA 15,255,128,7,255,128,7,255,0,7,255,128
1490 DATA 3,255,128,7,255,0,7,255,0,15,255,0
1500 DATA 15,255,0,15,255,0,3,255,0,3,254,0
1510 DATA 1,252,0,0,254,0,0,126,0,0,0,0,0,0
1520 DATA 0,0,0,0,0,0,0,0,0,0,0,0

```

LINE ANALYSIS FOR FLATCAT

- 1-5 The usual program header with title, credits, etc.
- 100-130 Clear and assign screen, color, sound, and other constants. DIMension string and integer arrays.
- 140 Clears screen; PRINT (white reverse field) title; cursor down twice. Prompts for instructions. PRINTs diamond graphic (pseudo-cursor) and backspaces cursor so the player's next keystroke will overlay the diamond.
- 150 POKEs in screen field and border colors. Places sprite #2 offscreen.
- 160 GETs player keystroke to continue game.
- 170 Turns all sprites off. Disallows irrelevant keystrokes.
- 180-260 Home the cursor; PRINT instructions: Set N = 0.

- 270 Increments N; gets a word from DATA. Puts each word into W\$(N) array; goes back for next word until "zymurgy" is read in.
- 280 Sets image pointer for sprite #2 to location 896 (= 14 × 64). READs in sprite image and POKEs into bytes starting at memory location 896.
- 290 Finishes loop above. Colors sprite #2 red; turns sprite #2 on.
- 300-400 Assign CT\$() the picture of the first cat; color him yellow; return to white at the end.
- 410-520 Assign CR\$() the picture of the cat with rock. Color him green; return to white.
- 530-650 Assign FC\$() the flatcat picture. Cursor down once at end of last line.
- 660 Randomizes the RND(d) function by using negative value for the dummy variable d. TI is an internal jiffy timer for the system clock.
- 670 Waits for any keystroke before continuing.
- 680 Increments game counter G; stops the program if all words are used up.
- 690 Zeroes the bad guess counter and selects next random word for play. If already used up then go get another one.
- 700 Clears the screen: X\$ is the new mystery word; nulls out W\$(R) so it cannot be chosen again.
- 710 Sets up L%(I) flag array for each letter of X\$; GETs each letter separately into the C\$() array; PRINTs a grey square in place of the mystery letters.
- 720-730 Set guessing limit and possible points for this word; prompt for a guessed letter.
- 740 PRINTs the player's guessed letter; PRINTs out cat #1.
- 750 GETs the player's next guess; disallows cursor key from messing up the display.
- 760 PUTs the guessed letters onto the screen in order so they won't be guessed again.
- 770 Sets a flag to "donk" the cat; makes sure keystroke is a letter of the alphabet; disallows other keystrokes.
- 780-790 Turn sound off; GET ASCII code of guessed letter; compare to each character in X\$; POKE into the grey squares where good letters go in the correct position; unset the donk flag if a good guess is made.

800-810 Go through each letter to see if they are all guessed; if so then the player won.

820 If the player muffs his last chance, then squishes the cat.

830 If the player makes a good guess then GOSUBs a small sound subroutine; jumps back for the next guessed letter.

840 Bad guess is assumed. Increments the guess counter; decrements points available. GOSUBs the small rock subroutine.

850 Cursor goes home and down three times; PRINTs score; goes back.

860 Sets the PRINT cursor down to the 8th line.

870-900 Player has gotten the word correctly. PRINT points; calculate overall percentage score; PRINT it out; prompt for next word.

910 GETs player keystroke to go to next word; puts sprite #2 (gory) offscreen; GOTO 680 to choose the next word.

920-940 Subroutine; drop a rock on the cat's head by screen POKEs; call the clonk subroutine.

950-980 Create clonk sound; PRINT cat #2; return from subroutine.

990-1000 Drop a heavy line to simulate the granite block.

1010-1030 PRINT in the flatcat (block); create squish sound.

1040 POKEs gory red sprite #2 beneath granite block; sends cursor home and PRINTs mystery word there.

1050-1060 Turn sound off; calculate and PRINT percentage score; go back to prompt for next mystery word.

1070-1080 Subroutine; make a small sound to reward correctly guessed letters.

1090-END DATA statements containing all possible words. Last word must be zymurgy because of line 270.

VARIABLE LIST FOR FLATCAT

In order of appearance:

SC and CS are the screen character and color memory locations.

QA = 198, a memory location counting keystrokes.

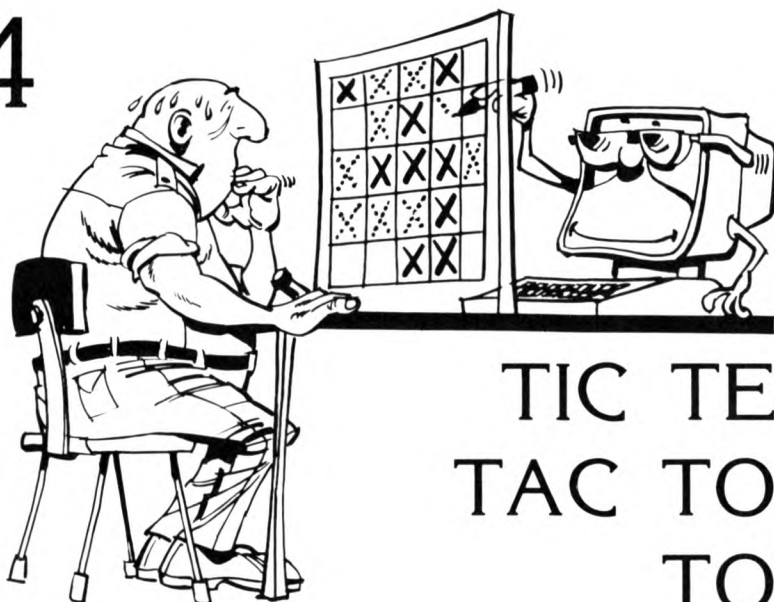
QB = 214, a memory location controlling which line to PRINT on.

VL is the memory location controlling sound volume.

C\$(n) is the nth letter of the word to be guessed.

P1 in memory controls the pitch of voice #1.
 P3 in memory controls the pitch of voice #3.
 AD in memory controls the attack/decay rates of voice #1.
 SR in memory controls the sustain/release of voice #1.
 WF in memory controls the waveform of voice #1.
 W\$(n) is the nth word we READ in from the DATA statements.
 CT\$(n) is the nth line of the first cat picture.
 CR\$(n) is the nth line of the "clonked" cat.
 FC\$(n) is the nth line of the Flatcat (granite block).
 L%(n) = 0 or 1; This flags whether the nth letter of the mystery word has been guessed and filled in.
 Y\$ is a player keystroke asking to continue the game.
 N is an index counter to READ each word into W\$(n).
 Z is a dummy variable used to randomize the RND function.
 Z\$ is a dummy keystroke to start play.
 G counts how many words have been played with.
 GS is how many bad guesses were made.
 GL is the limiting number of bad guesses permitted.
 R is a random number to select the random word W\$(R).
 X\$ = W\$(R), the word now being guessed.
 L is the length of the word being guessed in letters.
 C\$(n) is the nth letter of X\$ being guessed.
 SP = GL + 1; Possible points for the word being guessed.
 G\$ is the letter guessed by the player.
 D = 0 or 9; A flag signaling if the player has guessed all the letters.
 DK is a flag signaling a bad guess (and to donk the cat).
 SP is used to vary the pitch of the "donk" sound also.
 A and GA are the ASCII code for the player's guessed letters.
 TP is the total number of points earned.
 GT is the total points possible so far.
 PC is the percentage (score). $PC = 100\% \text{ times } TP/GT$.
 P is the screen position of the small rock as it is dropped on the cat in a subroutine.

4



TIC TEC TAC TOC TOE

Tic Tec Tac Toc Toe is a rudimentary attempt at artificial intelligence. The computer will try to beat the player in a game of extended tic tac toe and, failing that, will try for a stalemate. The game is played on a five-by-five board matrix, but only four markers in a row are needed to win. In the first game of any series, the machine will graciously offer the player the first move, and this initial advantage will be traded back and forth after that. Any horizontal, vertical, or diagonal row of four consecutive markers will win the game. The player's markers are double reverse field X's, while the computer uses double red squares.

When the player has the first move, it is possible to force a win, but you will want to discover how for yourself. The algorithm used in the program is a position-value or "greed" algorithm whereby the computer simply chooses the most valuable square when it is its turn. It is in assigning these values from all horizontal, vertical, and diagonal angles that takes up the bulk of the program listing as well as most of the computing time. Great care should be taken when typing in lines 420 through 1100 because they are so similar they invite mistakes. By allowing all possible diagonals, we create a lot of different situations for the machine to examine, each with a slightly different algorithm.

This is a good program to show to visitors since it demonstrates how a computer can pretend to be smart with a game that is very easily ex-

plained. As usual, the actual display on your screen will look better than the screen images shown in these pages.

The values used to increment squares in the subroutines from line 850 on determine the "personality" of the machine's play. If these values aren't carefully chosen, the machine will play too defensively or too aggressively and make blunders. The product S*9 influences the machine to pick its own winning move rather than to try to spoil a player's gambit if both happen at the same time. Feel free to experiment with these values, but if they are too far off, you will wind up with an exercise in artificial imbecility.

A4←				
A1	B1	C1	D1	E1
A2	⊗	⊗	⊗	⊗
A3	⊗	⊗	⊗	E3
⊗	⊗	⊗	⊗	E4
A5	⊗	C5	D5	E5

I WIN! NEW GAME?				
A1	B1	C1	⊗	E1
A2	⊗	⊗	⊗	⊗
A3	⊗	⊗	⊗	E3
⊗	⊗	⊗	⊗	E4
A5	⊗	⊗	D5	E5



YOU WON!

E 1

A1	B1	C1	D1	00
A2	B2	00	00	E2
A3	00	00	00	E3
A4	00	C4	00	E4
00	B5	C5	D5	E5

NOBODY WON!

E 2

00	00	00	00	00
00	00	00	00	00
A3	00	00	00	E3
00	00	00	00	00
00	00	00	00	00

LIST #4: TIC TEC TAC TOC TOE

```

1 REM*****
2 REM*TIC TEC TAC TOC TOE          10.20*
3 REM*LARRY HATCH                  *
5 REM*MENLO PARK CALIF 94025 1983 *
6 REM*****
100 CLR:PRINT "TIC TEC TAC TOC TOE":PRINT
110 QA=198:POKE53280,9:POKE53281,0:POKE53269,0
120 PRINT"TIC TAC TOE IS PLAYED HERE ON A "
130 PRINT"5 BY 5 MATRIX INSTEAD OF 3 BY 3."
140 PRINT:PRINT"FOUR PLACES IN A ROW WINS."
150 PRINT"DIAGONALS (8 POSSIBLE) ARE ALSO VALID."
160 PRINT:PRINT"IN ALL THERE ARE 28 WAYS TO WIN."
170 POKEQA,0:PRINT:PRINT"PRESS ANY KEY TO PLAY."
180 WAITQA,1:POKEQA,0:SC=1024:CS=55296
190 DIM B(6,6),L(28),V(6,6)
200 W=0:GOSUB 1190:RESTORE:SQ=0:W$=""
210 GM=GM+1:TN=2*(GM/2-INT(GM/2)):FORI=1TO5
220 FORJ=1TO5:B(I,J)=0:READ V(I,J):NEXTJ,I
230 IF TN=0 THEN PRINT"NOW I GO FIRST.":GOTO840
240 SC=1024:PRINT"
250 PRINT"YOUR MOVE: SQUARE? ◆ ■■";
260 POKE QA,0:WAITQA,1:GETA$:X=ASC(A$)-64
270 IF A$="R" THEN PRINT"RESIGNS":GOTO 200
280 IF X<10 OR X>5 THEN PRINT"? " :GOTO240
290 PRINTA$:POKEQA,0:WAITQA,1:GETY$:Y=VAL(Y$)
300 PRINTY$:IF Y<0 OR Y>5 THEN PRINT"?? " :GOTO250
310 XS=4*X-3:YS=12*Y+120:SD=SC+XS+YS:P=PEEK(SD)
320 IF P<10 OR P>5 THEN PRINTSPC(27)"? " :GOTO240
330 B(X,Y)=2:V(X,Y)=-1000:POKESD,214:POKESD+1,214
340 SQ=SQ+1:GOSUB1380:IF W=1 THEN PRINT"W$ WON!"
350 IF W=1 THEN POKEQA,0:WAITQA,1:GETA$:GOTO200
360 IF SQ<10 OR B(3,3)<>2 THEN 400
370 X=INT(3*RND(2)+2):Y=INT(3*RND(2)+2)
380 IF X=3 AND Y=3 THEN 370
390 GOTO1100
400 REM** VERTICAL TESTS **
410 FORI=1TO5:S=0:FORJ=1TO4:S=S+B(I,J):NEXTJ
420 IFS=6 OR S=15 THEN GOSUB 860
430 IFS=4 OR S=10 THEN GOSUB 870
440 S=0:FORJ=2TO5:S=S+B(I,J):NEXTJ
450 IFS=6 OR S=15 THEN GOSUB 880
460 IFS=4 OR S=10 THEN GOSUB 890
470 NEXTI:REM* HORIZONTAL TESTS *
480 FORJ=1TO5:S=0:FORI=1TO4:S=S+B(I,J):NEXTI
490 IFS=6 OR S=15 THEN GOSUB 900
500 IFS=4 OR S=10 THEN GOSUB 910
510 S=0:FORI=2TO5:S=S+B(I,J):NEXTI
520 IFS=6 OR S=15 THEN GOSUB 920
530 IFS=4 OR S=10 THEN GOSUB 930
540 NEXTJ:REM** DIAGONAL TESTS**
550 S=0:FOR K=1TO4:S=S+B(K,K):NEXTK
560 IFS=6 OR S=15 THEN GOSUB 940

```



```

570 IFS=4 OR S=10 THEN GOSUB 950
580 S=0:FOR K=1 TO 4: S=S+B(K+1,K):NEXT K
590 IFS=6 OR S=15 THEN GOSUB 960
600 IFS=4 OR S=10 THEN GOSUB 970
610 S=0:FOR K=1 TO 4: S=S+B(K,K+1):NEXT K
620 IFS=6 OR S=15 THEN GOSUB 980
630 IFS=4 OR S=10 THEN GOSUB 990
640 S=0:FOR K=2 TO 5: S=S+B(K,K):NEXT K
650 IFS=6 OR S=15 THEN GOSUB 1000
660 IFS=4 OR S=10 THEN GOSUB 1010
670 S=0:FOR K=1 TO 4: S=S+B(K,6-K):NEXT K
680 IFS=6 OR S=15 THEN GOSUB 1020
690 IFS=4 OR S=10 THEN GOSUB 1030
700 S=0:FOR K=1 TO 4: S=S+B(K,5-K):NEXT K
710 IFS=6 OR S=15 THEN GOSUB 1040
720 IFS=4 OR S=10 THEN GOSUB 1050
730 S=0:FOR K=2 TO 5: S=S+B(K,6-K):NEXT K
740 IFS=6 OR S=15 THEN GOSUB 1060
750 IFS=4 OR S=10 THEN GOSUB 1070
760 S=0:FOR K=2 TO 5: S=S+B(K,7-K):NEXT K
770 IFS=6 OR S=15 THEN GOSUB 1080
780 IFS=4 OR S=10 THEN GOSUB 1090
790 REM** PICK A STRATEGIC SQUARE **
800 VS=-1000:FOR I=1 TO 5:FOR J=1 TO 5
810 IF V(I,J)>VSTHEN VS=V(I,J):X=I:Y=J
820 NEXT J,I:GOTO 1100
830 REM** IF MACHINE GOES FIRST **
840 X=3:Y=3:GOTO 1100
850 REM* INCREMENT VALUES OF STRATEGIC SQUARES *
860 FOR J=1 TO 4:V(I,J)=V(I,J)+60+S*9:NEXT J:RETURN
870 FOR J=1 TO 4:V(I,J)=V(I,J)+30:NEXT J:RETURN
880 FOR J=2 TO 5:V(I,J)=V(I,J)+60+S*9:NEXT J:RETURN
890 FOR J=2 TO 5:V(I,J)=V(I,J)+30:NEXT J:RETURN
900 FOR I=1 TO 4:V(I,J)=V(I,J)+60+S*9:NEXT I:RETURN
910 FOR I=1 TO 4:V(I,J)=V(I,J)+30:NEXT I:RETURN
920 FOR I=2 TO 5:V(I,J)=V(I,J)+60+S*9:NEXT I:RETURN
930 FOR I=2 TO 5:V(I,J)=V(I,J)+30:NEXT I:RETURN
940 FOR K=1 TO 4:V(K,K)=V(K,K)+60+S*9:NEXT K:RETURN
950 FOR K=1 TO 4:V(K,K)=V(K,K)+30:NEXT K:RETURN
960 FOR K=1 TO 4:V(K+1,K)=V(K+1,K)+60+S*9:NEXT K:RETURN
970 FOR K=1 TO 4:V(K+1,K)=V(K+1,K)+25:NEXT K:RETURN
980 FOR K=1 TO 4:V(K,K+1)=V(K,K+1)+60+S*9:NEXT K:RETURN
990 FOR K=1 TO 4:V(K,K+1)=V(K,K+1)+25:NEXT K:RETURN
1000 FOR K=2 TO 5:V(K,K)=V(K,K)+60+S*9:NEXT K:RETURN
1010 FOR K=2 TO 5:V(K,K)=V(K,K)+30:NEXT K:RETURN
1020 FOR K=1 TO 4:V(K,6-K)=V(K,6-K)+60+S*9:NEXT K:RETURN
1030 FOR K=1 TO 4:V(K,6-K)=V(K,6-K)+30:NEXT K:RETURN
1040 FOR K=1 TO 4:V(K,5-K)=V(K,5-K)+60+S*9:NEXT K:RETURN
1050 FOR K=1 TO 4:V(K,5-K)=V(K,5-K)+25:NEXT K:RETURN
1060 FOR K=2 TO 5:V(K,6-K)=V(K,6-K)+60+S*9:NEXT K:RETURN
1070 FOR K=2 TO 5:V(K,6-K)=V(K,6-K)+25:NEXT K:RETURN
1080 FOR K=2 TO 5:V(K,7-K)=V(K,7-K)+60+S*9:NEXT K:RETURN
1090 FOR K=2 TO 5:V(K,7-K)=V(K,7-K)+25:NEXT K:RETURN
1100 REM*MACH BOARD MOVE

```

```

1110 POKESC+19,X:POKESC+20,Y+48:POKESC+21,31
1120 XS=4*X-3:YS=120*Y+120:SD=SC+XS+YS
1130 B(X,Y)=5:V(X,Y)=-1000
1140 POKE SD,102:POKESD+1,102:SQ=SQ+1
1150 SE=SD+CS-SC:POKESE,7:POKESE+1,7:GOSUB1380
1160 IFW$="I"THENPRINT"AI WIN! NEW GAME?"
1170 IFW$="I"THEN POKEQA,0:WAITQA,1:GOTO200
1180 GOTO240
1190 REM*BOARD SETUP
1200 PRINT"
1210 PRINT" A1 | B1 | C1 | D1 | E1"
1220 PRINT"
1230 PRINT"
1240 PRINT" A2 | B2 | C2 | D2 | E2"
1250 PRINT"
1260 PRINT"
1270 PRINT" A3 | B3 | C3 | D3 | E3"
1280 PRINT"
1290 PRINT"
1300 PRINT" A4 | B4 | C4 | D4 | E4"
1310 PRINT"
1320 PRINT"
1330 PRINT" A5 | B5 | C5 | D5 | E5"
1340 PRINT"
1350 RETURN
1360 DATA 9,8,7,8,9,8,11,11,11,8,7,11,16,11,7
1370 DATA 8,11,11,11,8,9,8,7,8,9
1380 REM** WIN EVALUATION **
1390 PRINT"
1400 L(I)=0:NEXTI:FORI=1TO4:FORJ=1TO5
1410 L(J)=L(J)+B(J,I):L(J+5)=L(J+5)+B(J,I+1)
1420 K=J+10:L(K)=L(K)+B(I,J):L(K+5)=L(K+5)+B(I+1,J)
1430 NEXTJ:L(21)=L(21)+B(I,I)
1440 L(22)=L(22)+B(I+1,I+1):L(23)=L(23)+B(I,6-I)
1450 L(24)=L(24)+B(I+1,5-I):L(25)=L(25)+B(I,5-I)
1460 L(26)=L(26)+B(I+1,6-I):L(27)=L(27)+B(I,I+1)
1470 L(28)=L(28)+B(I+1,I):NEXTI:W=0
1480 FORI=1TO28:IF L(I)=8THENW=1:W$="YOU"
1490 IFL(I)=20THENW=1:W$="I"
1500 IFW=0ANDSQ>21THENW=1:W$="NOBODY"
1510 NEXTI:RETURN

```

LINE ANALYSIS FOR TIC TEC TAC TOC TOE

- 1-5 Program header.
- 100-160 Clear all; clear screen; PRINT title in white; color screen and border; turn all sprites off; PRINT instructions.
- 170-200 Prompt and await keystroke; set screen and color constants: DIMension arrays; zero variables; set up gameboard.

- 210-220 Increment game counter; determine who gets the first move; zero out the squares-occupied array.
- 230 Branches for machine's turn to go first.
- 240-250 Clear old prompt and PRINT a new one.
- 260 GETs first keystroke of player's move; converts to ASCII.
- 270 Branches if player resigns from this game.
- 280 Disallows keystrokes out of range.
- 290 GETs 2nd keystroke of player's move.
- 300 Disallows keystrokes out of range.
- 310-320 Calculate screen location of player's move; disallow occupied positions.
- 330 Assigns occupancy of square to the player in B(m,n) array. Drastically devalues the square in V(m,n) so machine will disregard it; POKEs in player's markers.
- 340 Increments squares-occupied counter; GOSUBs win evaluation subroutine; announces any win.
- 350 If player won then WAITs for a keystroke and branches back.
- 360 Skips ahead if machine doesn't move first or the center square isn't occupied by player.
- 370 Otherwise, takes a random square for machine's first move.
- 380 Goes back if the machine randomly chooses center square, which is by now occupied.
- 390 GOTO routine POKEing in machine move; loops back to line 240.
- 400-460 Vertical tests; determine how many of whose men are on each vertical lane of four squares; GOSUB appropriate subroutine to increment values of the squares in each lane.
- 470-530 Horizontal tests; same as vertical test but lanes are left to right.
- 540-780 Examine each diagonal lane in turn as above in horizontal-vertical tests. GOSUB appropriate subroutines to increment values.
- 790-820 Find X,Y coordinate of highest-valued square using VS.
- 830-840 Grab center square if unoccupied.
- 850-1090 Increment all square values in the lane being examined, depending on who occupies some of those squares.

1100-1110 POKE announcement of the machine's move to top of screen.

1120-1150 Calculate screen memory location of machine's new squares; POKE in the characters; claim the position in the B(x,y) array; POKE in the color; increment the square counter; GOSUB win evaluation subroutine.

1160-1170 If player wins, announce win; branch back for next game.

1180 Goes back for player's next move.

1190-1350 Subroutine; set up playing board.

1360-1370 DATA; initial square values to start the game.

1380-1460 Win evaluation. Each of the 28 possible lanes is searched for five markers of a kind by adding the appropriate contents from the array B(n,m). These 28 sums are placed into the lane array L(1-28).

1480-1490 The L(I) array is examined for four men in a row. This determines the winner, if any.

1500-1510 If there is no winner and 21 or more squares are occupied, the game is declared a draw.

VARIABLE LIST FOR TIC TEC TAC TOC TOE

In order of appearance:

QA is the keystroke counter in the system scratchpad memory.

SC and CS are the first locations in screen character and color memory.

I, J, and K are general purpose index counters for FOR-NEXT loops.

B(I,J) records who occupies (claims) which squares on the board.

V(I,J) stores the present value of each of the 25 squares.

L(I) where I = 1 to 25 examines lines for winning situations.

W is a flag equal to 1 if somebody wins.

SQ is a counter. It counts squares that are occupied.

GM is a game counter. When even, the machine goes first.

TN is a zero if it is the machine's turn to go first.

A\$ GETs the first keystroke of the player's move.

X is for the Xth letter of the alphabet from 1 to 5.

Y\$ GETs the second keystroke of the player's move.

Y = 1 to 5 = the value of Y\$.

XS and YS are screen position coordinates.

SD = SC+XS+YS = where to POKE a player's or computer's markers on screen.

SE is a location in color memory to color a player's marker.

P = A screen PEEK to determine contents of a square for safety.

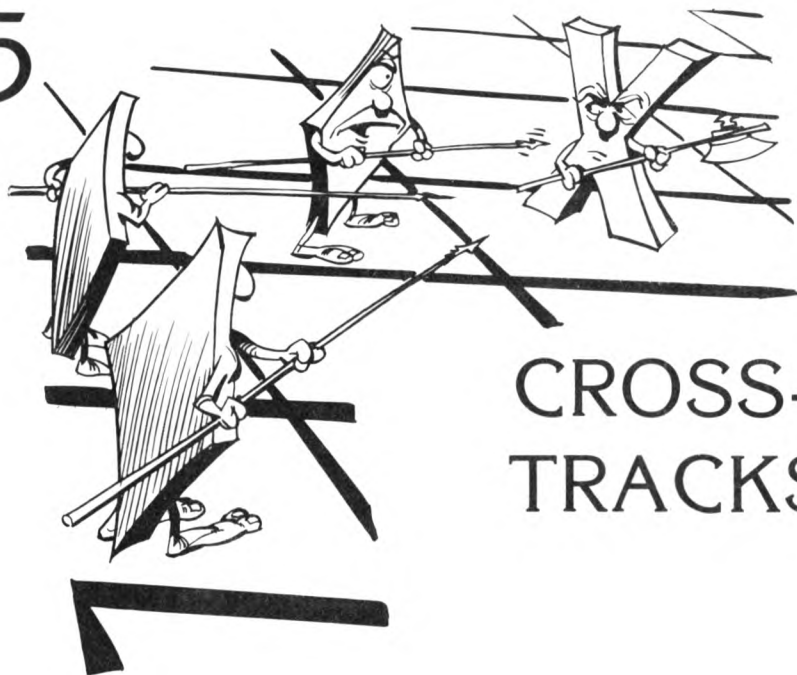
X and Y are variable coordinates for the computer's move.

S counts "claim numbers" in the B(I,J) array. Determines revaluation action to be taken.

W\$ says who won for PRINT statements at the end.

VS is the highest value found in the matrix so the machine can choose that square.

5



CROSS- TRACKS

Crosstracks is another tic-tac-toe type game sharing some similarities with program 4. The same square-value algorithm is used. In this game, the play is on a ten-by-ten board, and five in a row are needed to win. Once again, the player has the computer for an opponent, and this time, it is harder to beat the machine.

Unlike program 4, diagonal rows do not count. Winning rows are consecutive groups of five squares, horizontal or vertical, called *lanes*. Since there are six overlapping lanes in each row or column of ten squares, there are 120 lanes in which you can win or be beaten. I hesitate to say that there are 120 ways to win since the player has a choice of 8,136,038,400 different ways to place his first five markers alone! It will take several more moves than this to win if winning is even possible.

By eliminating the diagonal lanes, a tremendous amount of extra code in the form of IF-THEN statements and the like is removed. The purpose here is to get as much programming strength into as few lines as possible by using efficient algorithms. Only in this manner can enough "smarts" be programmed into a limited amount of memory space and time to even begin a serious initial study of artificial intelligence. One thing that must be done is to limit what the computer must do. The reader may want to program in diagonal lanes for himself later on.

The player always receives the first move. The machine rings a bell to call his attention, and that is the last favor the machine does for him. The

player enters his move by entering a letter from A-J for the row followed by a number from zero to nine for the column.

A diamond-shaped graphic character is POKEd to the chosen square, and the computer starts calculating its move immediately. While the machine examines the board, certain square numbers flash to the upper-left corner of the screen to let you know it is thinking. Sound effects also simulate machine thought. The row and column numbers are used to select pitches placed into voices one and three. These two pitches are logically ANDed by means of the synchronization waveform (POKE WF,19) to produce a most unexpected bassoon and flute duet. This duet begins with a rather modern harmony that gets progressively more avant-garde (or schizoid) as the gameboard squares are filled.

The machine will announce its move by printing a letter-number combination nearby and POKEing a large X to its chosen position. Another sound will play while the program searches all the lanes to see if anyone has won. This is brief at first, but it gets longer as more lanes fill up with markers.

Yet another subroutine plays a twisting scherzo when the machine checks to see if its last move won the game. This will only occur when one of the players had four in a row. It is fascinating to use the variables already present in an algorithm for sound generation. It costs little extra code and gives the machine a personality. At this point, the player's next move is signaled by a bell sound and a prompt and so on until the end of the game.

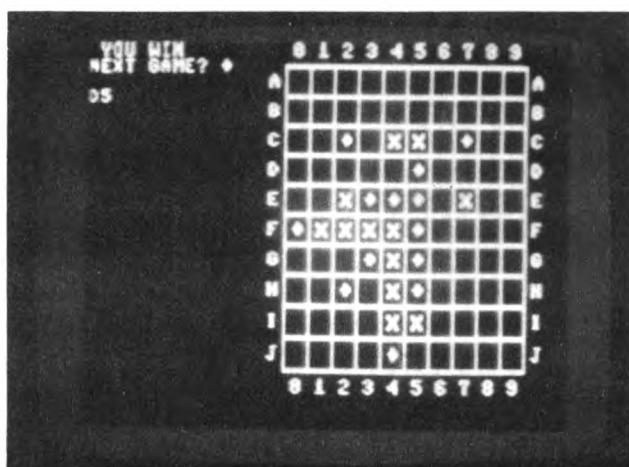
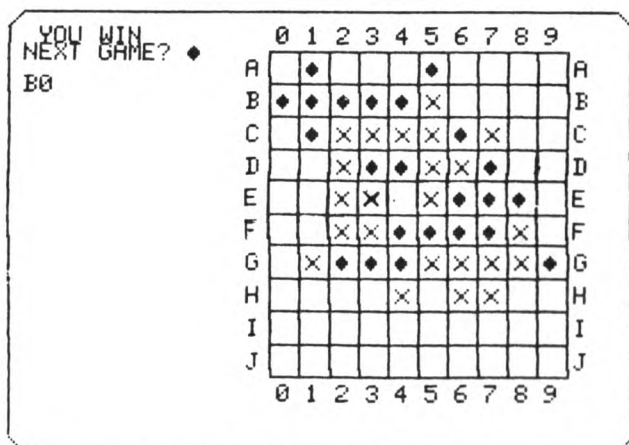
The computer's play is highly defensive at first as it tries only to block the obvious intentions of the player. After the squares have been repeatedly adjusted in value, the machine may try some aggressive moves. Once in a while, it will make an apparently nonsensical move that is much to its advantage later on! Sometimes, it makes a crazy move that is just that, crazy.

If the player is not paying attention, the computer may furtively build up an unstoppable position. If not, then play can proceed until no win is possible. It is up to the player to terminate a pointless match like this.

Any game may be considered "decided" when either opponent has four markers in a row with two open end squares. That opponent cannot be stopped from winning. For this reason, the computer will try to block all attempts to set up such positions. If the machine sees four markers in a row, even split in the middle, it will go into musico-logical frenzies. By careful choice of the values in lines 810 to 850, the computer will choose to advance its own gambit rather than to block the player's strategy when both are viewed as equal threats. It makes no sense to block a player's four-square threat when the machine can win immediately by completing its own four-square gambit. The squares in the center are given the highest initial values at the start of the game. They have the most room to expand in any

direction. From then on, play determines the updated values of each square. As in Tic Tec Tac Toe, the machine will select the square that it believes has the highest strategic value.

I don't want to make the game seem unbeatable; it certainly isn't. While the "greed" algorithm is very efficient for the amount of code in the program, the machine will eventually blunder its way into a loss. The most valuable square is not always the most important square, and no algorithm this simple can replace genuine thinking power.



YOU LOSE!
NEXT GAME? ♦
C3

	0	1	2	3	4	5	6	7	8	9	
A											A
B											B
C				X							C
D				X		X					D
E				X	♦	♦	X				E
F				X	X	♦					F
G				X		♦	♦				G
H					♦	♦	♦	♦			H
I							X				I
J											J
	0	1	2	3	4	5	6	7	8	9	

NOW YOU MOVE.
ROW (A-J) P
G0

	0	1	2	3	4	5	6	7	8	9	
A						X	♦	♦	♦	♦	A
B		X	X	X	♦	X	X	X	X	♦	B
C		♦	♦	X		♦	♦	♦	X	♦	C
D		X	X	♦	♦	♦	♦	X	X	♦	D
E		X	X	♦	X	♦	X	X	X	X	E
F		X	X	♦	♦	♦	♦	X	♦	X	F
G	X	♦	X	X	X	X	♦	♦		X	G
H	♦	♦	♦	X	♦	X	X	X	X	♦	H
I	♦	♦	♦	♦	X			X	X		I
J	♦	♦	♦	♦	X						J
	0	1	2	3	4	5	6	7	8	9	

LIST #5: CROSSTRACKS

```

1 REM*****
2 REM*CROSSTRACKS      COMMODORE 64 *
3 REM*BY LARRY HATCH   (C) 1983 *
4 REM*MENLO PARK, CA. 94025 V 10.21 *
5 REM*****
100 CLR:PRINT"  ** CROSS-TRACKS **"
110 QA=198:QB=214:SC=1024:CS=55296:POKE53280,9
120 POKE53281,11:POKE53269,0:VL=54296:P1=54273
130 P3=54287:AD=54277:SR=54278:WF=54276:POKEWF,0

```

```

140 DIM B(9,9),V(9,9):POKECS+40,1:POKECS+41,1
150 PRINT"INSTRUCTIONS? ♦♦":POKEQA,0:WAITQA,1
160 GETZ$:POKEQA,0:IFZ$<>"Y"THEN260
170 PRINT"THE OBJECT IS TO PLACE FIVE OF YOUR"
180 PRINT"MEN IN A ROW BEFORE THE COMPUTER DOES"
190 PRINT"LIKEWISE WITH ITS MEN.":PRINT
200 PRINT" ♦ ← YOUR MAN      X ← COMPUTER MAN."
210 PRINT"XDIAGONAL ROWS DO NOT COUNT.":PRINT
220 PRINT"PLACE YOUR MEN IN ANY BLANK SQUARE BY"
230 PRINT"ENTERING ITS LETTER THEN ITS NUMBER.X"
240 PRINT"PRESS ANY KEY TO PLAY. ♦♦";
250 POKEQA,0:WAITQA,1:GETZ$:PRINT$:POKEQA,0
260 PRINT"J":REM* BOARD SETUP *
270 PRINTSPC(15)"  0 1 2 3 4 5 6 7 8 9  "
280 PRINTSPC(15)"  | | | | | | | | | |  "
290 PRINTSPC(15)"  | | | | | | | | | |  "
300 FORI=1TO9
310 PRINTSPC(15)"  | | | | | | | | | |  "
320 PRINTSPC(15)"  | | | | | | | | | |  "
330 POKEVL,6:POKEVL,0:NEXTI:POKEWF,0
340 PRINTSPC(15)"  | | | | | | | | | |  "
350 PRINTSPC(15)"  0 1 2 3 4 5 6 7 8 9";
360 FORI=0TO9:L1=95+I*80:POKEESC+L1,I+1
370 POKECS+L1,1:L2=117+I*80:POKEESC+L2,I+1
380 POKECS+L2,1:NEXTI:REM* INITIAL SQUARE VALUES
390 FORI=0TO9:FORJ=0TO9:A=ABS(I-4.5):B=ABS(J-4.5)
400 V(I,J)=21-A-B:NEXTJ,I
410 FORI=0TO9:FORJ=0TO9:B(I,J)=0:NEXTJ,I
420 PRINT"NOW YOU MOVE.":PRINT"ROW (A-J) ♦♦";
430 POKEWF,0:POKEAD,12:POKEVL,15:POKEP3,31
440 WP=0:POKESR,0:POKEP1,133:POKEWF,21
450 POKEQA,0:WAITQA,1:GETR$:PRINTR$:R=ASC(R$)-65
460 POKECS+120,1:POKECS+121,1:IFR<00RR>9THEN420
470 POKECS+120,R+1:PRINT"XLINE(0-9) ♦♦";
480 POKEQA,0:WAITQA,1:GETL$:PRINTL$:L=VAL(L$)
490 IFASC(L$)<48ORASC(L$)>57THEN470
500 PRINT"X";R$L$ ← "":PRINT"
510 POKECS+121,L+48
520 IF B(L,R)<>0THENPRINT"X0000?? ":GOTO420
530 SD=SC+97+2*L+80*R:SE=CS+97+2*L:REM COLOR
540 POKE SD,90:POKESE,1:B(L,R)=1:V(L,R)=-9999
550 PRINT"XCOMPUTER MOVE ":PRINT" ←THINKING"
560 W=0:POKEWF,0:POKEVL,8:POKESR,250:REM VERTCL
570 JP=1:POKEP1,1:POKEP3,1:POKEWF,19:FORI=0TO9
580 POKEESC+40,I+1:FORJ=0TO5:S=0:FORK=JTOJ+4
590 S=S+B(I,K):NEXTK:GOSUB780:POKEESC+41,J+48
600 POKEP1,33+S*3:POKEP3,I+J:NEXTJ,I
610 JP=2:FORJ=0TO9:POKEESC+41,J+48:REM HORIZONTAL
620 FORI=0TO5:S=0:FORK=ITOI+4:S=S+B(K,J):NEXTK
630 INC=0:GOSUB780:POKEESC+40,I+1:POKEP3,I+J
640 POKEP1,40+S*3:NEXTI,J:POKEWF,0:IFW=0THEN690
650 IFW=1THENPRINT"X YOU WIN "
660 IFW=2THENPRINT"X YOU LOSE "
670 PRINT"NEXT GAME? ♦♦":POKEQA,0:WAITQA,1:GETZ$

```

```

680 PRINTZ$:RUN
690 VS=-9999:FORI=0TO9:POKEVL,9:POKEVL,0
700 FORJ=0TO9:IFV(I,J)>VSTHENVS=V(I,J):X=I:Y=J
710 NEXTJ,I:POKESC+120,Y+1:POKESC+121,X+48
720 SD=SC+97+2*X+80*Y:SE=SD-SC+CS
730 B(X,Y)=6:V(X,Y)=-9999:POKE SD,86:POKESE,1
740 IFWP=0THEN420:REM*← NEXT MOVE **
750 GOSUB900:IFW=2THENPRINT"  YOU LOSE !   "
760 IFW=2THEN670
770 GOT0420
780 INC=0:IFS=0THENRETURN:REM SQUARE INCREMENTS
790 IFS=5 THEN W=1:RETURN
800 IFS=30THEN W=2:RETURN
810 IFS=1 ORS=6THENINC=10:GOT0870
820 IFS=20RS=12THENINC=25+S:GOT0870
830 IFS=30RS=18THENINC=90+3*S:GOT0870
840 IFS=40RS=24THENINC=500+9*S:WP=1:GOT0870
850 IFS=70RS=80RS=90RS=140RS=19THENINC=-2
860 IF INC=0 THEN RETURN
870 ON JP GOTO 880,890
880 FORK=JTOJ+4:V(I,K)=V(I,K)+INC:NEXTK:RETURN
890 FORK=ITOI+4:V(K,J)=V(K,J)+INC:NEXTK:RETURN
900 PRINT"  CHECKING...":POKEWF,0:POKESR,250
910 POKEVL,8:POKEWF,21:FORI=0TO9:FORJ=0TO9:S=0
920 FORK=JTOJ+4:S=S+B(I,K):NEXTK:IFS=30THENW=2
930 POKEP3,I+J:POKEP1,48+S*3:NEXTJ,I:FORJ=0TO9
940 FORI=0TO9:S=0:FORK=ITOI+4:S=S+B(K,J):NEXTK
950 POKEP3,I+J:POKEP1,48+S*3:IFS=30THENW=2
960 NEXTI,J:POKEWF,0:RETURN

```

LINE ANALYSIS FOR CROSSTRACKS

- 1-5 The usual program header with title, credits, etc.
- 100 Clears all variables; clears screen; PRINTs (in white-reverse field) the title; ends reverse field.
- 120-130 Assign the usual keyboard, screen, color, and sound constants; turn sound off.
- 140 DIMensions arrays; POKEs color into upper-left screen corner for subsequent move displays.
- 150-160 Prompt WAIT and GET player keystroke for instructions; branch on keystroke.
- 170-240 PRINT instructions; reverse field "Q" sends cursor down one space. Backspace graphic after the diamond permits overprinting the diamond graphic pseudo-cursor.
- 250 GETs player keystroke to continue game.

260-350 PRINT 10 by 10 playing board to screen with a small sound effect.

360-380 POKE row letters (A-J) on either side of playboard.

390-400 Assign initial values to all squares on the playboard.

410-440 Declare all squares vacant in B(I,J) array; announce player's turn; ring a bell to wake him up.

450-470 GET player's first keystroke (A-J) for which row he chooses. POKE it to the upper-left screen corner.

470-490 GET player's second keystroke (0-9); POKE it next to his first one in the screen corner; disallow invalid keystroke.

500-510 PRINT player's square-name at screen top; blank out old prompts; POKE his second keystroke to the lower move display.

520 Disallows choice of an occupied space on the board.

530 Computes screen character and color memory locations.

540 POKES in player's marker and color; occupies B(I,J) array; heavily decrements value of chosen square so the machine won't try to choose it!

550-600 The computer's move. Zero the win flag W; POKE in sound variables; set jump vector for vertical tests; make "thinking" sounds; examine all vertical lanes of five squares each; GOSUB evaluation subroutine; POKE name of lane-top examined to top display; adjust sound pitches.

610-640 Set jump vector JP to 2 for horizontal tests; scan all horizontal lanes; POKE name of lane to top display; make more thinking sounds; turn sound off; test the win flag and branch ahead if no win.

650-680 Announce the presumed winner; prompt for next game; GET keystroke; RUN will start entire game from scratch!

690-710 Find the highest valued square in V(I,J) array; announce the choice in second screen display (three lines from top of screen).

720 Calculates memory locations for screen character and color to POKE in the machine's marker.

730-740 Claim the square; devalue the square; POKE in the machine's marker; go back if no win is possible.

750-770 GOSUB the win evaluation subroutine for machine win. (Possible player wins are tested between lines 550 and 640.) Announce any win and branch, usually to the player's next move at line 420.

780-890 is the lane evaluation subroutine, having six exit RETURNS!

780 Sends us right back for totally vacant lanes.

- 790-800 Set the win flag and return immediately if one is found.
- 810-850 Decide the value of the INCRement to add to each square in the lane being examined, positive or negative.
- 850 Detects mixed-occupancy lanes where nobody can win.
- 860 RETURNS immediately if no increment is decided. This line is a safety and should not execute. Lines 810-850 are believed to cover all contingencies.
- 870 Chooses one of two increment algorithms for horizontal or vertical lanes, based on the jump vector JP.
- 880 Increments all squares in a vertical lane by the value INC.
- 890 Increments all squares in a horizontal lane by INC.
- 900-960 The test for a machine win, after the machine moves. This subroutine is called only if the WP (win possible) flag is set.
- 900-910 Generate frantic sounds as if the machine is on to something.
- 910-930 Search vertically for a machine win.
- 930-960 Search horizontally for a machine win; make more sounds; set win flag as needed and RETURN.

VARIABLE LIST FOR CROSSTRACKS

In order of appearance:

- SC and CS are the screen character and color memory locations.
- QA = 198, a memory location counting keystrokes.
- QB = 214, a memory location controlling which line to PRINT on.
- VL, a memory location controlling sound volume.
- P1 in memory controls the pitch of voice #1.
- P3 in memory controls the pitch of voice #3.
- AD in memory controls the attack/decay rates of voice #1.
- SR in memory controls the sustain/release of voice #1.
- WF in memory controls the waveform of voice #1.
- B(I,J) is the square possession array, 10 by 10, where I indexes horizontally and J vertically. The contents of B(m,n) is zero for unoccupied squares on the gameboard; 1 if the player occupies and 6 if the machine occupies.
- V(I,J) is the continually updated value of each square on the board, as seen from the machine's point of view.

Z\$ is a player keystroke to continue play.

L1 and L2 are screen positions on either side of the playboard so the row letters (A-J) can be POKEd in.

I, J, and K are locally used index counters for FOR-NEXT loops.

A and B are used to set initial values into the V(I,J) array. Center squares are preferred, corner and edge squares having lower initial values.

R\$ is the letter name of a row so the player can choose a square.

R = 0-9 corresponding to letters A-J in R\$.

L\$ is the second keystroke (0-9)\$ for the player to choose his square.

L = VAL(L\$) is the number value of L\$; 0 to 9.

SD is the memory location on screen where we POKE in either the player's marker or the machine's marker.

SE is the memory location in screen color memory so the player's markers can be colored differently.

WP is the win possible flag. It is set to one if four markers of the same type are in a lane of five squares, the fifth being vacant.

W is the win flag. W=0 if nobody won yet. W=1 if the player wins and W=2 if the machine wins.

JP is a jump vector set to one during vertical tests and two during horizontal tests. Later, JP chooses one of two lines for an ON JP GOTO statement jump.

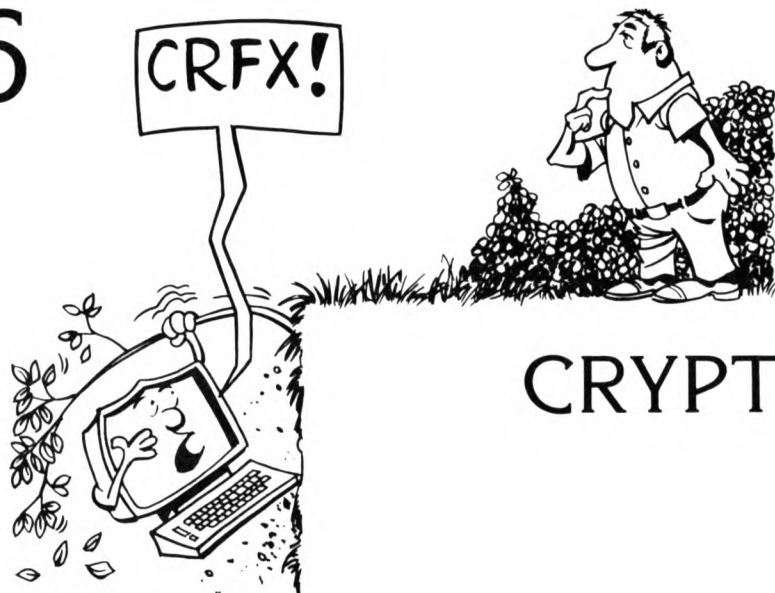
S is the sum of the five occupation numbers from any lane in the B(I,J) array. It indicates how many of whose markers are in the lane presently being examined.

VS is the highest value found in the V(I,J) as it is scanned for the best machine move.

X and Y store the coordinates (I,J) of the highest-valued square.

INC is a number chosen to increment or decrement all of the squares in a lane of five, based on how many of whose men already occupy that lane.

6



CRYPTO

This short program is intended for word puzzle enthusiasts. Crypto is a labor-saving program that keeps track of the letter substitutions temporarily made while trying to solve cryptograms. For the unfamiliar, a cryptogram is a message encoded by simple letter-for-letter substitution. Such a coding method is called the *Caesar cypher*, and I hope Caesar didn't try to use it for anything truly secret. These cyphers are so easily broken that amateurs can solve two or three of them over a cup of coffee. Cryptograms are found in daily newspapers, and most puzzle magazines have pages full of them.

A cryptogram is created when a plaintext message such as "Mary had a little lamb" is converted into "XRGN CRF R QVEEQU QRXT." R stands for the letter A throughout the message, and all other substitutions are equally consistent. It is up to the puzzle maniac to determine these substitutions and decipher back to the original message using pattern recognition and other skills.

You might immediately conclude that R is so common that it must stand for E. But that leaves us with a one-letter word E as the third word, which is absurd. Hypothesizing I or A will quickly lead to a solution in this case. The easiest cryptograms to solve are long ones with many short words, especially if the words are all very common.

Some cryptograms can be very difficult, however, and many tentative substitutions need to be made, each leading down a different path, before a solution suggests itself. This may take a lot of paper, and it can be quite

frustrating. Crypto will enable quick, accurate substitutions and a one-key reset capability to get out of blind alleys gracefully.

The first version of this program used string manipulation and input statements to INPUT the coded message. It was long and clumsy and has been completely rewritten. In the present version, the player enters the code exactly as it is printed in the magazine or newspaper. The screen is used as the memory rather than a series of strings. We need to allow for easy correction of the code as it is entered. After all, it's awfully easy to misspell CHPPAQIDIC.

We want to avoid the use of the RETURN key as much as possible on general principles, and certain keystrokes that are forbidden in string statements must be permitted. If a short message were to contain a comma and we enter it as, say, INPUT X\$(1), the computer will accept the portion up to the comma only. The error message EXTRA IGNORED will appear on the screen. The computer is saying, in effect, "I'm supposed to ask for one string input. You have sent me two, each separated by the usual BASIC comma delimiter!"

For these and other reasons, we must take total control of the input via GET statements and process the information to our specifications. This is pretty much the sort of method used in word processing programs to gain total control over typed input for storage onto disk as files. We store onto the screen for visibility and ease.

When the user makes an error, he can press the cursor left and delete keys to type over the mistake. At the end of a correctly displayed cryptogram, commas and all, he presses the asterisk key, which tells the machine that he is through with his entry.

Immediately, the program recreates a field of red squares, one for each letter, beneath the coded display. Spaces appear below corresponding to the spaces above. Commas and other punctuation marks appear below exactly as they appear above. It is now time to start decoding the message.

The machine prompts the player saying, "ENTER CODED LETTER". Using the "Mary had a little lamb" example, the player would enter an R. The machine then prompts "R BECOMES:" and the player enters A. Immediately, all of the red spaces in the lower display that correspond to R above turn into A's. The other red squares, the spaces, and the punctuation remain unaffected. Further substitutions lead to the solution.

If the player finds that he has made a muddle of substitutions leading nowhere, he simply presses the arrow-up key (above the RETURN key), and the original red field is restored for a new decoding strategy. I have found that decoding time on difficult cryptograms is cut at least in half by the simple expedient of having the computer keep track of these substitutions. It's more fun, too!

YEH BWO EDEMED HELXWREKER SFYD WK HWKX
 BE GPHSDM FHSBGLKMFKAPR KP CTFSD GPH
 FTT KXD SWLKFYDL?*

XXXXXXXXXXXXXXXXXXXX

ENTER CODED LETTER

YEH BWO EDEMED HELXWREKER SFYD WK HWKX
 BE GPHSDM FHSBGLKMFKAPR KP CTFSD GPH
 FTT KXD SWLKFYDL?*

AA NO PRAYER ITDOOPEA ETIS OF DEFA
 BA SPERY TEOOIFYTFOA FA MOTER SAY
 TOO FAR EOFTOR?*

ENTER CODED LETTER

YEH BWO EDEMED HELXWREKER SFYD WK HWKX
 BE GPHSDM FHSBGLKMFKAPR KP CTFSD GPH
 FTT KXD SWLKFYDL?*

HOW DID GEORGE WASHINGTON MAKE IT WITH
 NO FORMER ADMINISTRATION TO BLAME FOR
 ALL THE MISTAKES?*

ENTER CODED LETTER

LIST #6: CRYPTO

```

1 REM*****
2 REM* CRYPTO COMMODORE 64*
3 REM* HELPS WITH SIMPLE CRYPTOGRAMS*
4 REM* BY LARRY HATCH V 12.15*
5 REM* MENLO PARK, CA 94025 (C) 1983*
6 REM*****
100 CLR:PRINT"CRYPTOGRAM ORGANIZER"
110 QA=198:QB=214:SC=1024:CS=55296:CD=CS+480
120 POKE53269,0:POKE53280,9:POKE53281,15
130 PRINT"ENTER CODED TEXT VERBATIM, TYPE AN"
140 PRINT"ASTERISK '*' AT THE END OF THE CODE."
150 PRINT:PRINT"THE ARROW KEY (↑) WILL RESET THE"
160 PRINT"DECODING FIELD TO ALL RED SQUARES."
170 PRINT"PRESS ANY KEY TO CONTINUE.":POKEQA,0
180 WAITQA,1:POKEQA,0:PRINT:POKEQB,20:PRINT
190 PRINT"NOW ENTER CODE. END WITH AN ASTERISK"
200 PRINT:;N=0
210 POKEQA,0:WAITQA,1:GETC$:AS=ASC(C$)
220 IF AS=34 OR AS=17 THEN210
230 IF AS=20 THENPRINT:;N=N+1
240 PRINTC$:N=N+1:IFC$="*"THEN260
250 PRINT:;GOTO210
260 SD=SC+480:L=N+2:I=0:FORI=0TOL:C=PEEK(SC+I)
270 IFC>-1ANDC<27THENPOKE SD+I,102
280 POKECD+I,2:IFC>26THEN POKESD+I,C
290 NEXTI:POKEQB,20:PRINT
300 PRINT" — 37 — "
310 REM* GET NEXT SUBSTITUTION
320 POKEQB,19:PRINT:PRINT"ENTER CODED LETTER ";
330 POKEQA,0:WAITQA,1:GET X$
340 XX=ASC(X$):IFXX=94THEN260
350 IFXX=147THENPRINT:POKEQB,20:PRINT:GOTO190
360 IFXX<65ORXX>90THENPRINT"?":GOTO310
370 PRINTX$;
380 POKE QB,20:PRINT:PRINTX$" BECOMES: ";
390 POKEQA,0:WAITQA,1:GETY$:YY=ASC(Y$)
400 IF YY<65 OR YY>90 THENPRINT"?":GOTO380
410 PRINTY$:X=ASC(X$)-64:Y=ASC(Y$)-64
420 FORI=0TOL:W=PEEK(SC+I)
430 IFW=XTHENPOKESD+I,Y:POKECD+I,6
440 NEXTI:POKEQB,20:PRINT
450 PRINT" ";GOTO310

```

LINE ANALYSIS FOR CRYPTO

100 Clears variables; clears screen; PRINTs title in blue, reverse field; ends reverse field; cursor down.

110-120 Turn sprites off; assign usual PEEK and POKE locations for

color, screen, and keyboard; color border orange; color field light grey.

130-170 PRINTs instructions.

180 Keystroke to continue; PRINTs on screen line 21 next.

190 Prompts for coded input.

200 Sends cursor home; PRINTs first diamond-shaped pseudo-cursor; backspaces so player's next keystroke covers this cursor; zeroes the number of letters in the input.

210 GETs the next character of code; then its ASCII value.

220 Disallows quotation mark and cursor down keys; they mess up the whole screen display.

230 IF player presses the DELeTe key, THEN backspace; erases last character entered; decrements code-letter counter N.

240 Simply PRINTs the next character with a semicolon so they will all concatenate (string) together without spaces or dropping lines; increments letter counter; tests for asterisk to jump to decoding routine.

250 PRINTs the cursor after the last character entered; backspaces so it, too, can be written over by the next keystroke; goes back to line 210 for the next keystroke.

260 Assigns the first location (SD) for the dummy squares to be POKEd in: L = how many: PEEKs in screen characters from original code, one by one.

270-280 If screen character is alphabetic, it is mirrored below by a red square. If not alphabetic, then copy it exactly (spaces and punctuation).

290 Ends loop above; PRINTs on screen line 21 next.

300 PRINTs 38 spaces followed by cursor-home symbol. Clears off old prompts. Sends real cursor home.

310 Self-explanatory REMark; is not executed.

320 Prompts for next substitution on screen line 21.

330 GETs the letter to be transformed.

340 Checks for up-arrow key. This calls for a fresh start with the same original code.

350 Checks for SHIFTEd CLR key; a signal to start from scratch! The original cryptogram is lost.

360 Disallows non-alphabetic characters from substitution.

370 Shows the letter selected for substitution.

380-390 GET hypothesized plaintext letter Y\$, that X\$ stands for.
 400 Disallows characters not in the alphabet.
 410 Displays Y\$; converts X\$ and Y\$ into screen code; X,Y.
 420-440 Go through entire code input on the screen; substitute supposed plaintext letter into corresponding position in the red boxes; set screen line 21 for next PRINT line.
 450 Blanks out the old prompt on line 21; goes back for the next substitution.

VARIABLE LIST FOR CRYPTO

In order of appearance:

QA = 198 is the keystroke counter memory location.
 QB = 214 is the register controlling which screen line to print on.
 SC is the first location in screen character memory.
 CS is the first location in screen color memory.
 CD = CS + 480 is where the red squares will start.
 C\$ = each character of the original cryptogram in turn.
 AS = the ASCII value of each C\$ from the GET statements.
 N = a running count of how long the cryptogram is in characters.
 SD = SC + 480, the first location of the red squares in screen character memory. Contrast with CD.
 L = N+2 = how many red squares to POKE into SD+ and CD+.
 C = values PEEKed from first screen display memory SC+(0 to L).
 I is an ordinary index counter in FOR-NEXT loops.
 X\$ is the coded letter you want to change from the code display.
 Y\$ is what you change X\$ into for the red display below.
 XX is the ASCII value of X\$ to disallow odd characters.
 YY is the ASCII value of Y\$ to permit the alphabet only.
 W is temporary storage used to compare code with $X = ACS(X\$) - 64$.
 X and Y are XX and YY minus 64 to permit POKEing the correct character in screen character code. This code differs from Commodore ASCII. ASCII THEM why! Commodore ASCII differs from standard ASCII as well.

7



KENO

Keno is a very old game. Over 2000 years ago, it was invented by the Chinese to help finance an internal power struggle at the end of the Chou dynasty. The forces financed by this novel scheme reunited China under the Han dynasty by 206 B.C. The game flourishes to the present day. In the mid 1800s, Chinese immigrants brought the game to the United States, and soon all nationalities were playing it. In the West, it became known as Chinese Lottery.

By 1934, the game appeared (legally) in Reno, Nevada, as Race Horse Keno and, finally, just Keno. By the early 1960s, the game was pretty well standardized in its present form. Keno is probably the second biggest money maker for the Nevada casinos, after slot machines. A \$1 or \$2 ticket can win up to \$25,000, so there is no lack of players, night or day.

In the casino, a player marks numbers onto a blank ticket that has squares numbered from 1 to 80. The most popular tickets are marked with about eight spots and cost almost \$1. After marking, say, eight spots, the player hands the ticket to a dealer in a booth who makes a duplicate ticket, keeps the original ticket (and the dollar, of course), and gives him the duplicate as a receipt. Drawings are held every few minutes.

All over the casino, large boards light up the 20 numbers that are randomly chosen from a wire basket holding 80 numbered ping-pong balls. No, I'm not making this all up. If enough of the numbers that you origi-

nally marked are among the 20 that are chosen, you win. Payoffs range from a few dollars to about \$18,000 for "catching" all your spots.

A study of probability theory neatly categorizes the odds of winning in a formula called "hypergeometric distribution." This is a fancy term for pulling colored balls from an urn. Solutions to this formula indicate that your chances of winning a big ticket are so remote that the casinos should pay more than they do. House advantage, over the long run, is approximately 30%. This can soar to 38% or more in tourist-oriented Nevada casinos.

This program will first ask the size of the ticket, from two to nine spots. Having selected, say, eight, you are asked to key in these numbers (between one and eighty.) For one-digit numbers, just press the number and RETURN or other key. Use the E key to blank out errors. You can use it repeatedly. Try it. When all numbers are in, the computer will construct a Keno board on your screen. It then underlines the numbers you have chosen.

The game number and your running "bank" are displayed at the top of the screen. Your numbers are sorted numerically and are displayed at the top. The computer then randomly generates 20 different numbers and lights up the board by putting those numbers into "reverse field," i.e., solid yellow on black. As the numbers appear, sounds are heard to mimic the croupier calling out. The numbers chosen appear at the lower left of the display.

When the 20 numbers are all lit, the computer compares your ticket automatically and prints the results at the bottom of the screen. If you win anything, it is added to your bank at the top of the screen. Your bank starts with a balance of zero so you can see if you ever get ahead of the house.

At the end of the game, you are given three options: R for Replay, which plays the same ticket again; N for new card (you enter different numbers and play resumes); and W for wheel (the present ticket is played over and over, endlessly). You have to use the STOP key to prevent the computer from gambling on and on.

Other features will become apparent. I would not suggest any alterations until the original LIST is correctly running. Watch the spaces, color, and cursor control characters carefully when typing them in.

All tickets cost \$1. Payouts are as follows:

- 2-SPOT: 2 pays \$12 (not \$16!)
- 3-SPOT: 2 pays \$1, 3 pays \$42
- 4-SPOT: 2 pays \$1, 3 pays \$42, 4 pays \$110
- 5-SPOT: 3 pays \$2, 4 pays \$20, 5 pays \$485

- 6-SPOT: 3 pays \$1, 4 pays \$4, 5 pays \$85, 6 pays \$1570
 7-SPOT: 4 pays \$2, 5 pays \$21, 6 pays \$320, 7 pays \$5,000
 8-SPOT: 5 pays \$8, 6 pays \$85, 7 pays \$1640, 8 pays \$18,000
 9-SPOT: 5 pays \$3, 6 pays \$42, 7 pays \$280, 8 pays \$4000, 9 pays \$18,000

These payouts are roughly averaged from typical Nevada Keno rate-sheets away from the Las Vegas strip and other tourist traps. Federal law requires that winnings of \$1800 or more, on a \$1 ticket, be reported to the IRS. Your Commodore 64 is probably exempt from such rulings.

This program is published for the third time here. It has been continually updated and improved since 1978, when I wrote my first version on a Commodore PET®. This version is written especially and exclusively for the Commodore 64®.

21	22	23	24	31	32	33	34	44	
2 YOUR BANK = \$-2									
1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80

21	22	23	24	31	32	33	34	44
----	----	----	----	----	----	----	----	----

1 YOUR BANK = \$-1
 1 2 3 4 5 6 7 8 9 10
 11 12 13 14 15 16 17 18 19 20
 21 22 23 24 25 26 27 28 29 30
 31 32 33 34 35 36 37 38 39 40
 41 42 43 44 45 46 47 48 49 50
 51 52 53 54 55 56 57 58 59 60
 61 62 63 64 65 66 67 68 69 70
 71 72 73 74 75 76 77 78 79 80
 MATCHES 21 23 33 34
 FOR 4 NUMBERS
 3=REPLAY, 4=NEW CARD, 5=WHEEL;
 ENTER CHOICE *



21	22	23	24	31	32	33	34	44
----	----	----	----	----	----	----	----	----

52 YOUR BANK = \$-52
 1 2 3 4 5 6 7 8 9 10
 11 12 13 14 15 16 17 18 19 20
 21 22 23 24 25 26 27 28 29 30
 31 32 33 34 35 36 37 38 39 40
 41 42 43 44 45 46 47 48 49 50
 51 52 53 54 55 56 57 58 59 60
 61 62 63 64 65 66 67 68 69 70
 71 72 73 74 75 76 77 78 79 80
 MATCHES 21 22 24 31 44
 3=REPLAY, 4=NEW CARD, 5=WHEEL;
 ENTER CHOICE *

21	22	23	24	31	32	33	34	44
83 YOUR BANK = \$-80								
1	2	3	4	5	6	7	8	9
11	12	13	14	15	16	17	18	19
21	22	23	24	25	26	27	28	29
31	32	33	34	35	36	37	38	39
41	42	43	44	45	46	47	48	49
51	52	53	54	55	56	57	58	59
61	62	63	64	65	66	67	68	69
71	72	73	74	75	76	77	78	79
MATCHES 21 22 23 24 31								
PAYOUT = \$ 3.00								

LIST #7: KENO

```

1 REM*****
2 REM*2-9 SPOT KENO      COMMODORE 64 *
3 REM*BY LARRY HATCH    (C) 1983 *
4 REM*MENLO PARK, CALIF 94025 12.15 *
5 REM*****
100 CLR: DIM A(9), B%(80), D(20), P%(9,9): QA=198
110 QB=214: P1=54273: WF=54276: AD=54277: SR=54278
120 VL=54296: SC=1024: CS=55296: POKE53281,0
130 PRINT"***** 2-9 SPOT KENO *****": POKEQB,21
140 PRINT:PRINT" BY LARRY HATCH (C) 1983"
150 FORI=1TO9:FORJ=0TO9:READP%(I,J):NEXTJ,I
160 DATA 0,3,0,0,0,0,0,0,0,0,12,0,0,0,0,0,0
170 DATA 0,0,1,42,0,0,0,0,0,0,0,1,4,110,0,0,0
180 DATA 0,0,0,0,0,2,20,485,0,0,0,0,0,0,0,0,0
190 DATA 0,0,0,1,4,85,1570,0,0,0,0,0,0,0,0,0,0
200 DATA 0,0,0,0,2,21,320,5000,0,0,0,0,0,0,0,0,0
210 DATA 0,0,0,0,0,8,85,1640,18000,0,0,0,0,0,0,0,0
220 DATA 0,0,0,0,0,3,42,280,4000,18000,0,0,0,0,0,0,0
230 PRINT"*****HOW MANY SPOTS TO MARK (2-9) *****";
240 POKEQA,0:WAITQA,1:GETT$:PRINTT$:TK=VAL(T$)
250 IFTK<2ORTK>9THEN230
260 PRINT"*****NOW ENTER YOUR"TK"LUCKY NUMBERS"
270 QI=TI:U=0:POKEQB,21:PRINT
280 PRINT" <TO ERASE ERRORS HIT THE 'E' KEY> "
290 FORI=0TO119:POKESC+I,32:NEXTI:FORI=1 TO TK
300 PRINT"*****":POKEQA,0:IFU>0THEN410
310 PRINT"*****":WAITQA,1:GETA$:PRINT"*****":A$;
320 POKEQA,0:WAITQA,1:IFA$="E"ANDI=1THEN300
330 IFA$="E"THENI=I-2:PRINT"*****TAB(I*3+1)"XX"
340 IFA$="E"THEN410

```

```

350 GETB$: IFB$="0" THEN A(I)=10*VAL(A$):GOTO390
360 IFB$="E" THEN PRINT "*****":POKEQA,0:GOTO310
370 IFVAL(B$)=0 THEN A(I)=VAL(A$):GOTO390
380 A(I)=10*VAL(A$)+VAL(B$)
390 PRINT "  ";A(I):IF A(I)<1 OR A(I)>80 THEN U=2
400 PRINT "  ":PRINT TAB(I*3-3)A(I)"  "
410 PRINT "  ":NEXT I:I=0
420 QQ=INT(TI-QI):REM** RIPPLE SORT FOLLOWS **
430 FOR S=1TOTK-1:P=0:FORQ=1TO(TK-S)
440 IF A(Q)<A(Q+1) GOTO 480
450 IF A(Q)=A(Q+1) THEN U=1
460 IF A(Q)<1 OR A(Q)>80 THEN U=2
470 X=A(Q):A(Q)=A(Q+1):A(Q+1)=X:P=1
480 NEXT Q:IF P=0 THEN S=TK-1
490 NEXT S
500 IFU=1 THEN PRINT "  DONT USE SAME NUMBER TWICE"
510 IFU=2 THEN PRINT "  NUMBERS FROM 1 TO 80 ONLY!"
520 IFU>0 THEN 230:REM* INPUT ERROR
530 PRINT "  ";
540 H=F:FOR I=1TOTK:PRINTA(I):NEXT I:G=INT(G+1)
550 BK=INT(BK-.99):GOSUB950:GOSUB1000
560 REM -BALLS REPLACED IN DRUM-
570 FOR E=1TO80:B%(E)=E:NEXT E:F3=1
580 PRINT "*****GAME #";G;" YOUR BANK =$"BK"  "
590 F=0:H=0:V=0:REM* RANDOM BALLS CHOSEN
600 SC=1024:SD=1182:FOR T=1TO20
610 F=INT((800*RND(F3)+F2)+2)
620 IF F<81 GOTO 640
630 F=F-80:GOTO620
640 W=F:F3=INT(VAL(TI$)/119)+QQ
650 F2=W:IF B%(W)=0 GOTO 610
660 H=B%(W):D(T)=H:B%(W)=0
670 TN=INT((H-1)/10):UN=H-10*TN:SE=SD+UN*4+TN*80
680 LF=PEEK(SE-1):RT=PEEK(SE):POKESE-1,LF+128
690 POKESE,RT+128:POKE QB,19:PRINT:PRINT"  "
700 PT=20+H/2:LD=8:GOSUB1050:NEXT T
710 C=0:I=0:K=0:REM*COMPARATOR
720 POKEQB,19:PRINT:PRINT"MATCHES ";
730 FOR I=1TOTK:K=A(I)
740 IFK=0 THEN NEXTI
750 POKEVL,12:FORL=1TO20:PT=C*8+10
760 IF K=D(L) THEN C=C+1:TP=50:GOSUB1050:PRINTK;
770 NEXT L,I:M=0:REM** ANY PAYOUT? **
780 POKEQB,20:PRINT:IF C>0 THEN 800
790 PRINT "*** NONE **",:GOTO850
800 M=P%(TK,C):IF M=0 GOTO 820
810 PRINT "*****WINNER!  -$"M"1.00":GOTO830
820 PRINT "FOR"C"NUMBERS":GOTO850
830 BK=BK+M:REM* PAYOUT AND BRANCHING
840 PRINT "*****GAME #";G;" YOUR BANK =$"BK"  "
850 POKEQB,21:PRINT:IF N$="W" THEN 930
860 PRINT "R=REPLAY; N=NEW CARD; W=WHEEL; "
870 PRINT "ENTER CHOICE  " :POKEQA,0:WAITQA,1
880 GETN$: IFN$<>"R" ANDN$<>"N" ANDN$<>"W" THEN 880

```

```

890 PRINTN$; IF N$="R" THEN 940
900 IF N$="W" THEN T1=4: GOTO 930
910 IF N$="N" THEN 230
920 IF T1=0 THEN 940: REM** TIME DELAY **
930 FOR Y1=1 TO (T1*150): Y=1: Y=0: NEXT Y1
940 PRINT "X": GOTO 540: REM* REPLAY
950 PRINT "XXXXX": REM* BOARD SETUP
960 L$=" 1 2 3 4 5 6 7 8 9 10X"
970 PRINTL$: A=1: AB=2: B=10: FORI=ATO7: FORK=ATOB
980 PRINT K+B*I; TAB(K+AB): NEXTK: PRINT: NEXTI
990 FORI=1823 TO 2023: POKEI, 32: NEXTI: RETURN
1000 REM* UNDERLINE CHOSEN NUMBERS
1010 SD=1222: FORI=1 TO TK: H=A(I): TN=INT((H-1)/10)
1020 UN=H-10*TN: SE=SD+UN*4+TN*80: SF=SE-SC+CS
1030 POKESF-1, 1: POKESF, 1: POKESF-41, 3: POKESF-40, 3
1040 POKESE-1, 69: POKESE, 69: NEXTI: RETURN
1050 POKEP1, PT: POKEAD, 32: POKESR, 240: POKEVL, LD
1060 POKEWF, 17: FORZ=1 TO 5: POKEP1, PT+Z: NEXTZ
1070 POKEWF, 16: RETURN
1080 RETURN

```

LINE ANALYSIS FOR KENO

100-120 Clear variables; DIMension arrays; assign keyboard, screen, color, and sound constants; POKE the screen-field black.

130-140 Clear screen; PRINT title in yellow; set screen line 21 for PRINTing credits. 150-220 READ in payout array from the DATA up to line 220.

230-250 GET a keystroke for how many spots on the ticket; disallow irrelevant keystrokes.

260-290 Prompt for the n lucky numbers chosen; POKE out old prompts at top of screen.

300 Prompts for two-keystroke entry with double pseudo-cursor; tests for previous entry error.

310 GETs a keystroke; PRINTs it over the righthand diamond.

320 GETs the next keystroke; tests if first keystroke is E for error; branches to re-enter first keystroke.

330 Tests for error in subsequent numbers to decrement index counter and XX out the nth (bad) entry only.

340 Branches back if the above error is detected.

350 GETs the second keystroke; tests if it is the zero number key; branches ahead if so.

- 360 Tests for E = error on the player's second keystroke in any given two-digit entry; branches back to re-enter both keystrokes if so.
- 370 Tests for non-numeric second keystroke; if so, then first keystroke represents a one-digit number so jump ahead.
- 380 Two-digit number assumed; multiplies first keystroke by ten; adds second keystroke.
- 390 PRINTs the Ith lucky number; sets error flag U if it's out of range.
- 400 Places the Ith lucky number in the Ith place in a neat row.
- 410 Blanks out the diamond pseudo-cursor and completes this convoluted loop.
- 420 Counts the QQ jiffies it took to enter all n numbers above. A jiffy is 1/60 second.
- 430-490 Ripple-sort routine. This puts the chosen lucky numbers into numerical order.
- 500-520 Test the U-flag for errors in number entry; announce the error and send the player back to re-enter them.
- 530-540 Clear the screen, PRINT out the player's lucky numbers in sorted numerical order; increment game counter for next drawing.
- 550 Decrements and rounds off the player's bank by a dollar; calls subroutines to set up the board and to underline the player's chosen numbers on the board. Nevada won't do that.
- 560-570 Reset the B%(80) board numbers.
- 580-590 Next drawing; PRINT statistics; zero variables.
- 600-700 Loop to choose 20 random balls; update the board on-screen, etc.
- 610-630 Merry-go-round randomizer for better random distribution of the balls chosen. Counteracts biases in the Commodore RND function.
- 640-650 Process randomizing variables; go back if the chosen ball is already removed from the drum.
- 660 H = new number drawn; remove it from the "drum" B(); put the number in the "drawn" array D().
- 670-690 Determine screen location of drawn number on the field; put it into reverse field to light it up; POKE it back onto the board.
- 700 Sets pitch and volume values; GOSUBs sound; ends ball-drawing loop.

- 710-720 Zero variables for comparison of ticket to the balls drawn; prepare to PRINT them on screen bottom.
- 730-740 I-loop will index through the player's lucky numbers disallowing invalid (zero) entries.
- 750-770 L-loop indexes through the 20 drawn numbers; catch-counter C increments if any numbers match; sounds made, etc.
- 780 Skips ahead to 800 if any numbers were caught.
- 790 PRINTs "none" and jumps ahead.
- 800 Determines the prize, if any, from the P%(TK,C) array; skips ahead if prize is nothing.
- 810 Announces winner; skips to 840.
- 820-830 Shows how many numbers were caught (matched up).
- 840 Increments the player's bank by the prize; PRINTs statistics.
- 850 Prepares to print on 22nd line; if in wheel mode, skips ahead.
- 860-880 Ask for and GET player's next option keystroke. Be insistent.
- 890-910 Branch according to player option chosen; set flag if player chooses to "wheel" the game.
- 920 Skips over time delay routine if not in wheel mode.
- 930 Time delay to read results on screen (may be adjusted).
- 940 Goes back for the next play of the same ticket in replay or wheel mode.
- 950-990 Subroutine; set up the Keno board on the screen; POKE out all the old prompts on screen bottom.
- 1000-1040 Subroutine; underline the player's lucky numbers on the Keno board for his n-spot ticket.
- 1050-1080 Subroutine; make some sort of sound.

VARIABLE LIST FOR KENO

In order of appearance:

QA and QB are the usual keyboard and screen line control locations.
A(n) is the array of n lucky numbers chosen by the player.
B%(1-80) is the playboard array. B%(n) = n unless number n comes up in the draw; then B%(n) = 0.

D(20) is the array that will hold the 20 numbers drawn from 80.

P%(9,9) is the payoff array. The prize = $P\%(sp, sd)$ where sp is how many spots were marked on your ticket, and sd is how many of those spots were subsequently drawn.

P1, AD, SR, VL, and WF are the usual sound-generation memory locations. SC and CS are the usual screen character and color memory locations.

T\$ and TK are the size of the ticket in spots, 2 to 9.

QI, QQ, and TI help to randomize the RND function more thoroughly.

U is a flag that signals lucky-number entry errors; out of range, etc.

A\$ and B\$ are the two keystrokes to enter a lucky number. For single-digit numbers, follow the digit with a space or return key.

E, I, L, Q, S, and T are used locally as index counters in FOR-NEXT loops.

S, P, Q, and X are used in the ripple-sort routine, to put the player's lucky numbers into numerical order.

H is the next number drawn from the basket, each ball in turn.

F is a large number used in a merry-go-round randomizing scheme to help select better distributions of the random numbers H.

G counts games. One per drawing.

BK is the player's bank. He starts with a bank of zero.

F2 and F3 help to randomize the RND function.

TN and UN determine the X,Y coordinates of the number H on the screen.

LF and RT PEEK in the contents of one place on the board so it can be highlighted in reverse-field. This is done by adding 128 to these contents and POKEing them right back.

SE is the character memory location of RT (and hence LF) on the screen.

PT is a pitch, passed to the sound-generating subroutine.

LD is the loudness (0 to 15) POKEd into VL (volume) in the sound subroutine.

K is each player's number, taken in turn to compare with the board after drawing.

C is the count of numbers "caught" or matching the 20 drawn numbers on the board.

M is the final payout, or prize, if any, in dollars.

N\$ is the keystroke selecting a new ticket, a replay of the old ticket, or a non-stop wheel of the present ticket.

T1 is a time delay constant so the player can study the board in the wheel mode. It is also the flag indicating this mode. If $T1 = 0$, the play will stop at the end of the present game.

Y and Y1 are dummy variables in the time delay loop described above. A, AB, and B are assigned numerical values only to speed up the board setup subroutine.

SF is the color memory location of the positions on-screen where the underlines occur, to highlight selected numbers.



There you are, minding your own business in the local tavern, reading your sandwich and eating the newspaper, when the UFOs attack again. This time, the wimpy little aliens are throwing antimatter pebbles. They are trying to annihilate the beer!!

Being an electrical engineer of sorts, you know what to do. You get the well-used battery jumper cables from the trunk of your car and make a Tesla coil out of the rotating television antenna on top of the Oasis Beer Garden. Now you can strike back with ball lightning!

Ball lightning has been observed and noted by scientists from the days of ancient Greece to the present. The very idea of ball lightning invites skepticism, and that's hardly surprising. (It seems like a dandy weapon to shoot at flying saucers, however.) Niccola Tesla, a Yugoslav-American scientist and inventor was probably also skeptical. His Tesla coils were designed to be a means of transmitting not only information (i.e., radio) but also large amounts of electrical power through the air.

Experimenting in a Colorado laboratory, and using very large amounts of power, he accidentally created observable ball lightning. He also burnt out the generators at the local power company. (For an exciting history of those years, may I recommend the book, *TESLA: A Man Out of Time*, by Margaret Chaney [Laurel/Dell Publishing, 1981]. It reads like science fiction. It isn't.)

In this game, the aliens zero in on your Tesla coil. One direct hit and the game is over. Their fuel is their ammunition: antimatter. Nothing has more energy than antimatter. Not even hydrogen fusion can offer the enormous energy of matter-antimatter annihilation. The aliens can't shoot with laser-like speed. Instead they fling small pieces of antimatter from a magnetic gun that cannot physically touch the stuff. Likewise, ball lightning moves at a relatively lazy pace. For this reason, both you and the aliens have to aim ahead at your target.

It is curious to note how similar ball lightning observations are to certain (hypothetical) passages of tiny amounts of antimatter through our atmosphere. Both would appear as intense localizations of energy. Both would create a plasma by superheating and superenergizing the atmosphere around them. One could only guess that they were identical if there were some way to explain how they seem unaffected by the effects of gravity imposed by regular matter. If that were the case, a random visit from an infinitesimal piece of antimatter might be indistinguishable from a gob of so-called ball lightning.

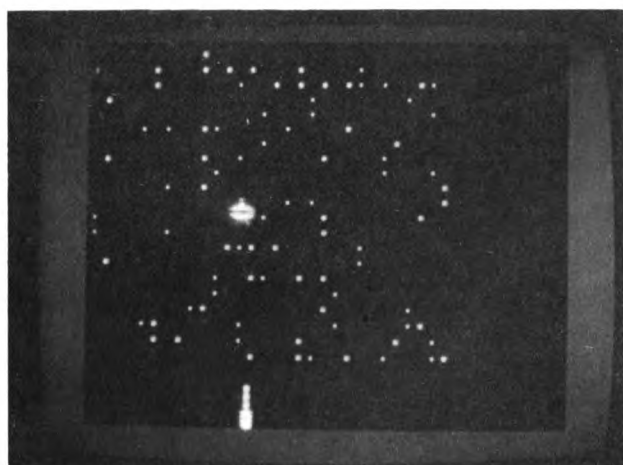
This wild speculation may serve to temporarily explain some of the more disastrous effects of ball lightning. When it does touch something, that something is destroyed.

Reflecting on the similarities between the two phenomena, I decided to use the same sprite image (differently colored) for the ball lightning and antimatter missiles.

The sound effects include ring modulation, synchronization, and the noise waveform for explosions. If the Tesla coil is hit, we flash changes in the screen color to mimic the great release of energy. Zapped saucers run through their sprite colors for the same reason, and they disappear.

There is no upper limit to the shots you can fire and no theoretical limit to the scores. If a game ends with an annihilated Tesla coil, the all-time high score is displayed along with the present score for comparison. You may want to program in a shots-fired limit to bring the game to a more timely end. This would best be done immediately after line 360 by incrementing a shot-counter of your own design. The two cursor keys control left and right motion of the Tesla coil, while any letter key will fire the ball lightning.

YOU ARE DEFENDING THE OASIS BEER
 GARDEN FROM UFOS. SCORE POINTS BY
 SHOOTING THEM WITH BALL LIGHTNING.
 THE SAUCERS DROP SMALL PARTICLES OF
 ANTIMATTER. ONE HIT WILL ANNIHILATE
 THE WHOLE PLACE, BEER AND ALL.
 MOVE LEFT AND RIGHT BY USING THE
 TWO CURSOR KEYS AT BOTTOM-RIGHT.
 PRESS ANY KEY TO PLAY ♦



LIST #8: SAUCERS

```

1 REM*****
2 REM*SAUCERS!      FOR COMMODORE 64 *
3 REM*BY LARRY HATCH      (C) 1983 *
4 REM*MENLO PARK, CA 94025      11.3 *
5 REM*****
100 PRINT"  "      22 FLYING SAUCER ATTACK! 22"
110 CLR:SC=1024:CS=55296:POKE53280,9:POKE53281,0
  
```

```

120 PRINT"  YOU ARE DEFENDING THE OASIS BEER"
130 PRINT"  GARDEN FROM UFOS.  SCORE POINTS BY"
140 PRINT"  SHOOTING THEM WITH BALL LIGHTNING."
150 PRINT"  THE SAUCERS DROP SMALL PARTICLES OF"
160 PRINT"  ANTIMATTER.  ONE HIT WILL ANNIHILATE"
170 PRINT"  THE WHOLE PLACE, BEER AND ALL."
180 PRINT"  MOVE LEFT AND RIGHT BY USING THE"
190 PRINT"  TWO CURSOR KEYS AT BOTTOM-RIGHT."
200 P1=54273:P3=54287:AD=54277:VL=54296:WF=54276
210 SX=53248:SY=53249:PX=53250:PY=53251:FX=53252
220 FY=53253:AX=53254:AY=53255:CL=53278:PS=0
230 RG=53269:POKERG,0:FORI=0TO3:POKE2040+I,13+I
240 POKE53287+I,1:NEXTI:POKE2043,15:QA=198
250 FORI=0TO190:READD:POKE832+I,D:NEXTI:HS=0
260 PRINT"  PRESS ANY KEY TO PLAY  ♦":POKEQA,0
270 WAITQA,1:PRINT"  ␣":FORI=0TO100:QZ=21*RN(1)
280 RZ=30*RN(1):SZ=RZ+40*QZ:POKESCS+SZ,46
290 POKECS+SZ,1:NEXTI:POKE53290,3:POKEQA,0:ZF=""
300 SV=50:FV=230:PV=230:SH=20:FH=150:PH=150:DR=2
310 AV=50:AH=20:POKESY,SV:POKEFY,FV:POKEPY,PV
320 FB=0:POKESX,SH:POKEFX,FH:POKEPX,PH:POKERG,7
325 PRINT"  SCORE"TAB(30)"  "
330 GETZF:IFZF=""THEN370
340 A=ASC(ZF):IFA=17THENPH=PH-3:IFPH<15THENPH=255
350 IFA=29THENPH=PH+3:IFPH>255THENPH=15
360 IFA>30THENPOKEWF,20:POKEAD,29:FB=1:POKEWF,21
370 POKEPX,PH:IFFB=1THENPOKEP1,FV:GOTO390
380 FH=PH:FV=PV:POKEFX,FH:POKEFY,FV:GOTO510
390 FV=FV-4:IFFV<40THENFV=PV:FH=PH:FB=0:A=0
400 POKEFY,FV:POKEFX,FH:IFPEEK(CL)=5THEN420
410 GOTO510:REM* SKIP OVER BALL-SAUCCER COLLISION
420 FB=0:PS=PS+1:POKEWF,0:POKEP1,35:POKEAD,7
430 POKEVL,15:FORI=0TO15:POKEWF,129:POKE53287,1
440 POKE53289,15-I:POKEWF,0:NEXTI:FH=PH:FV=PV
450 SV=30:PRINT"  SCORE"TAB(30)"  "
460 POKEFX,FH:POKEFY,FV:POKE53287,1:POKE53289,1
470 RZ=6*RN(1):SV=50+20*RZ:VS=2+INT(3*RN(1))
480 RZ=2*RN(1)+1:ONRZ GOTO 490,500
490 SH=255:DR=-VS:GOTO510:REM* SAUCER LEFT
500 SH=20:DR=VS:POKEP3,SV:REM* SAUCER RIGHT
510 SH=SH+DR:IFSH>253ORSH<20THEN470
520 POKESX,SH:POKESY,SV:IFAM=1THEN570:ANTIMATTER
530 R=99*RN(1):IFR>2THEN330:REM* NO ANTIMATTER
540 AV=SV+15:AH=SH:DX=PH-SH:DY=PV-SV:IX=DX/35
550 IY=DY/35:POKERG,15:POKEAX,AH:POKEYA,AV:AM=1
560 POKEWF,0:POKEAD,29:POKEVL,8:POKEWF,19:GOTO330
570 AH=AH+IX:AV=AV+IY:POKEAX,AH:POKEYA,AV
580 POKECL,0:POKEP1,SH:POKEP3,AH:IFAV<240THEN610
590 POKE53281,1:AM=0:POKERG,7:POKEWF,0:POKESR,7
600 POKEP1,40:POKEWF,129:POKE53281,0:GOTO330
610 P=PEEK(CL)AND10:IFP<10THEN330
620 AM=0:POKEWF,0:POKEP1,30:POKEAD,7:POKEVL,15
630 FORI=0TO15:POKEWF,129:POKE53281,15-I
640 POKEWF,129:NEXTI:POKERG,1:AH=255:AV=0
650 POKEAX,AH:POKEYA,AV:POKEQA,0:IFPS>HSTHENHS=PS
660 PRINT"  HIGH"TAB(30)"  "

```


450-460 PRINT score; POKE ball and Tesla sprites home; restore their original colors.

470-500 Next UFO; assign random speed direction and altitude with saucer.

510-520 Advance UFO on its path; go back for new saucer if it goes off screen.

530 Randomly decides to fire a tiny piece of antimatter (a.m.).

540-560 Place a.m. just beneath the saucer; calculate trajectory in IX, IY increments to position; turn this sprite on along with the others in RG memory location; turn AM flag on; set sound variables; go back for next player move if any.

570-600 Move any pre-existing antimatter down; increment a.m. co-ordinates; POKE these in; clear collision register!! Update sound pitches; branch ahead if still on screen.

590-600 Antimatter went off screen; turn screen field white; make a bang; screen goes black again; go back for next player keystroke.

610 Tests for antimatter-Tesla collision; branches back for next keystroke if not.

620-650 Tesla annihilated; AM flag is zeroed; a loop takes the entire screen through 15 colors; sounds of explosion; RG set so only the saucer remains; antimatter POKEd into limbo (off screen); high score raised if necessary.

660-680 End of game; statistics, prompts, etc.; branch back for a new game.

690-730 64 bytes defining the UFO-saucer image; includes a dummy zero.

740-790 The Tesla coil sprite image. This is the player's game piece.

800-830 Ball sprite image. This is used for both the ball lightning and the antimatter sprites. The two sprites have their image pointers aimed at the same 63 byte image.

VARIABLE LIST FOR SAUCERS

In order of appearance:

SC and CS are the first locations in screen character and color memory.

P1 and P3 are the memory locations of the pitches of voices #1 and #3.

AD and WF are the memory locations for attack/decay and waveform for voice #1.

VL is the memory location for volume on all voices in the sound chip. SX, SY are memory locations for the X, Y coordinates of the UFO sprite.

PX, PY are memory locations for the X, Y coordinates of the player (Tesla coil).

FX, FY are memory locations for the X, Y coordinates of the ball lightning sprite (fireball).

AX, AY are memory locations for the antimatter X, Y coordinates.

CL is the memory location of the sprite collision register. Any sprite involved in a wreck has its own bit turned to a one in this byte of memory.

PS is the player's score; one for each saucer zapped.

RG is the memory location of the sprite on-off register; one bit turns on its particular sprite.

QA is the keystroke counter memory location in scratchpad memory.

D is a variable used only to call in the sprite image numbers from DATA statements.

HS is the highest score yet played in a given playing session.

Q%, R%, and S% are integer variables used to fill screen space with stars randomly.

SV, SH are the saucer's vertical and horizontal coordinates on the screen.

FV, FH are the fireball's coordinates (x, y).

PV, PH are the player's (Tesla coil) sprite screen coordinates. All of these coordinates are in pixels.

AV, AH are the antimatter sprite coordinates.

DR is a signed (positive or negative) velocity for the antimatter sprite.

FB is a flag. FB = 1 when the ball lightning is flying.

Z\$ is a one-keystroke string used to GET the player's next move.

A is ASCII code for Z\$ so we can manipulate it.

VS is the absolute (unsigned) speed of the UFO in motion.

R is a random variable to fire antimatter at unexpected times.

DX, DY are the X and Y differences between the saucer coordinates and the Tesla coordinates.

IX, IY are the increments to be added to the antimatter coordinates to aim it right at the Tesla coil.

AM is a flag that indicates antimatter in motion when set to one.



BLACKJACK

There are a number of blackjack versions and programs in existence. In Great Britain, a very similar game is called Twist. This program is based upon the game as played in most Nevada casinos. The deal remains with the house, in this case the computer. Dealer does not take pushes or hands of equal value; these are declared a standoff, and no money changes hands. The one major deviation is that lack of screen space and a desire for a short program prevent us from splitting hands. This may be a blessing in disguise. The author once watched another player repeatedly splitting pairs of tens and queens. He must have been a tourist.

Another oddity is that the dealer deals the first card to himself. Since the cards are randomly drawn, this doesn't affect the play.

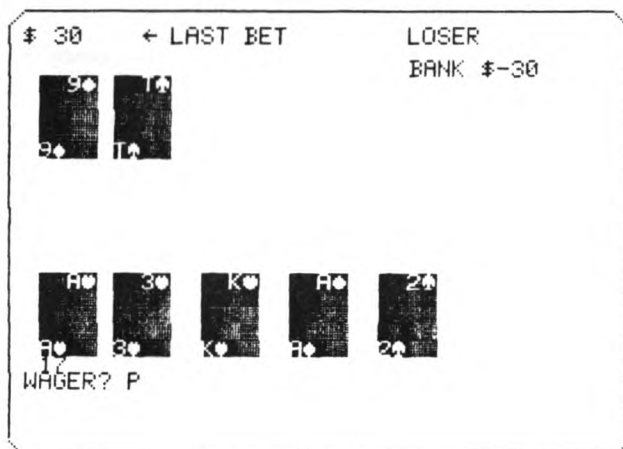
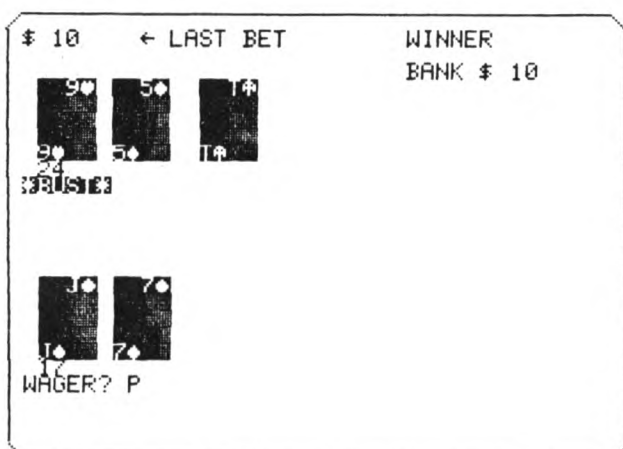
Otherwise, play proceeds as usual. The player is allowed to double down on all hands of 11 or less, which is slightly non-standard, but it is in the house's interest to permit this. If a player wants to double down with a hand of four playing against me, I will certainly let him do so. In doubling down, the player doubles his bet and receives one and only one card face down. That is his final hand.

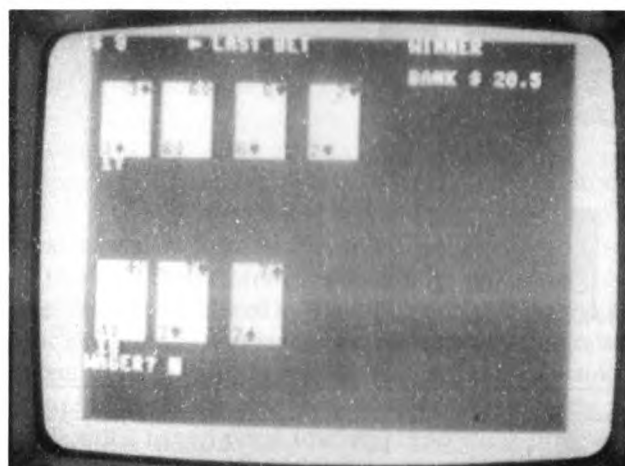
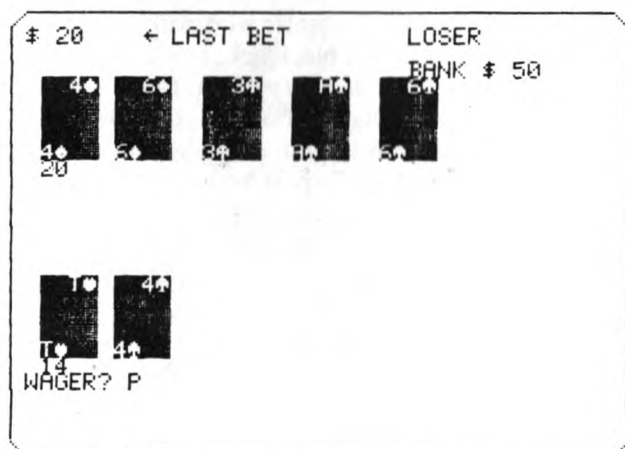
Assuming the player is standing pat, the dealer must hit all hands below 17. He must even hit (or twist) a soft 17, which contains a high-ace reducible to a low ace of one point. A blackjack is defined as any two-card hand totalling 21 points. This must contain an ace (high) and a face card or ten. When the player gets a blackjack, the dealer pays him one and a

half times his original wager. If the dealer wins with a blackjack, only the original wager is lost. If both get a blackjack, it is again a standoff.

This program, like Keno, is densely written. I'm not implying that the writer is dense. What I mean is that a lot of things happen in relatively few lines; that the program loops into and out of itself in an apparently tangled manner. Because of this, you should type it in exactly according to the LIST before attempting any alterations or improvements.

One improvement would be to take control of the player's moves and give them to a subprogram. Such a subprogram could be an automatic betting system of your own design. You might borrow some ideas from a mathematician named Edwin O. Thorp. He wrote the book *Beat the Dealer* (Vintage Books, 1966). He is barred from entering virtually every casino in Nevada.





LIST #9: BLACKJACK

```

1 REM*****
3 REM* BLACKJACK FOR COMMODORE 64 *
4 REM* BY LARRY HATCH 11.4 *
5 REM* MENLO PARK, CA 94025 1983 *
6 REM*****
100 CLR:PRINT" ";:SC=1024:QB=198:QB=214
110 J$="BLACKJACK":B$="BUST"
120 POKE53280,14:POKE53281,11:POKE53269,0
130 P1=54273:P3=54287:AD=54277:WF=54276:VL=54296

```

```

140 PRINT"BLACK-JACK!!!":GOSUB 750
150 PRINT"YOUR INITIAL BANK IS ZERO TO HELP KEEP"
160 PRINT"TRACK OF WINNINGS OR LOSSES. YOU CAN"
170 PRINT"BET FROM $1 TO $1000 PER HAND."
180 PRINT"PRESS S TO STAND PAT AND H TO HIT."
190 PRINT"PRESS D TO DOUBLE-DOWN WITH HANDS OF"
200 PRINT"ELEVEN OR LESS. PRESS Q TO QUIT."
210 PRINT:PRINT"YOU CANNOT SPLIT PAIRS, SORRY."
220 W=0:POKEQB,20:FORI=840TO856:POKESC+I,32:NEXTI
230 POKE54278,0:PRINT:INPUT"WAGER";U$
240 L$=LEFT$(U$,1):IFL$="N"ORL$="Q"THEN END
250 W=VAL(U$):IFW=0THEN220
260 PRINT"W:POKEQA,0:POKEQB,20:PRINT
270 IFW>1000THENPRINT"$1000 IS MAXIMUM":GOTO220
280 B=B-W:V=W:LP=1:GOSUB790:REM DEAL CARDS NOW
290 DP=1:CP=0:GOSUB 890:DV=VC:DA=AA:DH=VC
300 GOSUB 790:PH=VC:PA=AA:DP=2:CP=0:GOSUB 890
310 PRINTPH:GOSUB790:X$=D$:DH=DH+VC:DA=DA+AA:DP=1
320 CP=1:GOSUB 950:GOSUB 790:DP=2:CP=1:GOSUB 890
330 PH=PH+VC:PA=PA+AA
340 IFPH>21ANDCP=1THENPRINT J$:PJ=1:GOTO470
350 IFPH>21 ANDPA=0THENPRINTB$:PH:PB=1:GOTO590
360 IF PH>21 AND PA>0 THEN PH=PH-10:PA=PA-10
370 IF DD=1 GOTO 470
380 PRINTPH:REM ** HIT/STAND OPTIONS
390 PRINT"HIT OR STAND?♦":POKEQA,0:WAITQA,1:ZD=0
400 GETH$:PRINT"♦H$:IFH$="D"ANDPH>11THENZD=1
410 IFZD=1THENPRINT"CANT DOUBLE!♦":GOTO390
420 IF H$="S" THEN 470:REM* THE PLAYER STANDS PAT
430 IF H$="H" THEN 460:REM* PLAYER HITS HIS HAND
440 IF H$="D"THEN DD=1:GOTO 730:REM* DOUBLES DOWN
450 PRINT"♦":GOTO390:REM* GO BACK FOR GOOD INPUT
460 CP=CP+1:LP=CP:GOSUB790:GOSUB890:GOTO330
470 D$=X$:DP=1:CP=1:GOSUB890:CP=2
480 IFDH=21THENDJ=1:GOTO560:REM* DEALER BLACKJACK
490 IF DH=22 AND DA>0 THENDH=DH-10:DA=DA-10
500 IF DH>18 ANDDH<22 THEN560:REM* DEALER STANDS
510 IF DH=17 ANDDA=0 GOTO560:REM* DEALERS HARD 17
520 GOSUB790:DH=DH+VC:DA=DA+AA:GOSUB890:PRINTDH
530 IFDH>21ANDDA=0THENPRINT B$:DB=1:GOTO560
540 IF DH>21 AND DA>0 THEN DH=DH-10:DA=DA-10
550 CP=CP+1:GOTO 500
560 IF DD<>1 GOTO 590
570 CP=LP+1:DP=2:D$=W$:GOSUB 890:PRINTPH
580 IF PH>21THEN PB=1:REM* THE PLAYERS HAND BUSTS
590 IF PB=1THEN W=0:GOSUB1040:GOTO660
600 IF DJ=1ANDPJ<>1THEN W=0:GOTO 660
610 IF PJ=1 AND DJ<>1 THEN W=W*2.5:GOTO 660
620 IF PJ=1 AND DJ=1 GOTO 660
630 IF DB=1 THEN W=W*2:GOSUB1040:GOTO 660
640 IF PH>DH THEN W=W*2:GOTO660
650 IF DH>PH THEN W=0:GOSUB1010:GOTO660
660 PB=0:DB=0:PJ=0:DJ=0:DD=0:PRINT"♦";SPC(26);
670 IF W=V THENPRINT"STANDOFF"

```

```

680 IF W=0 THENPRINT"LOSER"
690 IF W>V THENPRINT"WINNER"
700 B=B+W:W=0:V=0:PRINT:PRINTSPC(26)"BANK $"B
710 PRINT" ";SPC(8); "← LAST BET":DH=0:DA=0:AA=0
720 PH=0:PA=0:V=0:PRINT" ";GOTO 220
730 GOSUB 790:W$=D$:CP=LP+1:REM* DOUBLE DOWN CARD
740 DP=2:GOSUB950:B=B-W:W=W*2:V=V*2:GOTO 330
750 C$="A♠2♠3♠4♠5♠6♠7♠8♠9♠T♠J♠Q♠K♠":REM MAKE DECK
760 C$=C$+"A♥2♥3♥4♥5♥6♥7♥8♥9♥T♥J♥Q♥K♥"
770 C$=C$+"A♦2♦3♦4♦5♦6♦7♦8♦9♦T♦J♦Q♦K♦"
780 C$=C$+"A♣2♣3♣4♣5♣6♣7♣8♣9♣T♣J♣Q♣K♣":RETURN
790 REM ** PULL 1 CARD AND SHRINK THE DECK **
800 R=2*INT(LEN(C$)*RND(LEN(C$))/2+1)-1
810 N$=MID$(C$,R,1):Y$=MID$(C$,R+1,1)
820 T$="":IF R>1THENT$=LEFT$(C$,R-1)
830 C$=T$+MID$(C$,R+2):D$=N$+Y$:AA=0:C=LEN(C$)
840 IFC=10THENPRINTSPC(26)"NEW DEAL":GOSUB750
850 VC=VAL(N$):IF VC<>0 GOTO 880
860 IF N$="A" THEN VC=11:AA=10:GOTO880
870 IFN$="T"ORN$="J"ORN$="Q"ORN$="K"THENVC=10
880 RETURN:REM** PRINT CARD FACE UP ROUTINE BELOW
890 PKEWF,0:POKEP1,DP*10+ASC(D$):POKEAD,25
900 PRINT" ";IFDP=2THENPRINT"XXXXXXXXXX":PRINT
910 PKEWF,17:FORK=1TO(CP*6):PRINT" ";NEXTK
920 PRINT" ";D$;"XXXXXX"XXXXXX"XXXXXX";
930 PRINT"XXXXXXXXXX"D$" ";POKEWF,0
940 RETURN:REM ** DOWN CARD SUBR. BELOW
950 PKEWF,0:POKEVL,10:POKEP1,CP*6+10*DP
960 PKEAD,9:POKEP3,DP+ASC(D$)/2:POKEWF,21
970 PRINT" ";IF DP=2 THENPRINT"XXXXXXXXXX"
980 FORK=1TO(CP*6):PRINT" ";NEXTK
990 PRINT"XXXXXXXXXXXXXXXXXXXXXXXXXXXX";
1000 PRINT"XXXXXXXXXX":RETURN
1010 POKEVL,10:POKEP1,7:POKE54275,15:POKEAD,63
1020 PKEWF,129:FORZ=10TO50:POKEP1,Z/2:NEXTZ
1030 PKEWF,0:RETURN:REM* 1SLURP. BUST BELOW.
1040 POKEP1,40:POKEAD,8:POKEVL,15:POKEWF,129
1050 FORZ=0TO50:NEXTZ:POKEWF,0:RETURN

```

LINE ANALYSIS FOR BLACKJACK

- 100-110 Clear variables; assign screen keyboard and string constants.
- 120 POKes the border blue, screen field grey; turns sprites off.
- 130-140 Assign sound memory location constants; PRINT title; set up the deck of cards as a long string.
- 150-210 PRINT the game instructions.

- 220-230 Beginning of the next hand of Blackjack. POKE sustain/release off just in case; prompt for next wager.
- 240-250 Allow player to resign; disallow irrelevant keystrokes.
- 260-270 PRINT wager at screen upper left; impose \$1000 house limit.
- 280-290 Dealer takes the first card!
- 300-310 Player's first card dealt, then dealer's down card.
- 320-330 Player's second card is dealt.
- 340-460 Player's plays his hand; it is evaluated.
- 340 Tests for immediate player's blackjack.
- 350 Tests if the player has busted.
- 360 Drops an ace from 11 to 1 point to keep player under 22.
- 370 Branches to a routine to get a third card (down) if player doubles down.
- 380-410 GET player's next option; disallow doubling down on a hand greater than 11.
- 420-440 Branch out depending on the option the player chooses.
- 450-460 Disallow irrelevant keystrokes; PRINT hit card face up if chosen; loop back.
- 470-560 Player stands; dealer plays his hand.
- 470 Dealer turns up his hole card, which was face down.
- 480-510 Test action for dealer's double aces, standing pat, and standing with a hard 17.
- 520 Dealer hits his own hand.
- 530 Tests for dealer's bust.
- 540 Drops an ace to keep dealer under 22.
- 550 Branches back for the next dealer card if any.
- 560 Jumps ahead to 590 if the player didn't double down.
- 570-580 Turn up the player's double down card; test again if the player busted.
- 590-720 Moment of truth; the hand is decided.
- 590 Ends game if player has already busted.
- 600-610 Test for exclusive blackjacks, player or dealer; branch.
- 620 Tests if both have a blackjack; branches.
- 630-640 Test if dealer busted or player has the better hand.
- 650 Tests if the dealer's hand is higher.
- 660 Zeroes all the test variables; moves cursor to screen top right.

- 670-690 PRINT the decision at the cursor position determined above.
- 700-720 Pay the wager W (W will be zero if you lost); PRINT statistics; zero more variables; go back for next hand of Blackjack.
- 730-740 Routine; player doubles down. His bet, W, and its comparator V are doubled. The card is printed face down; jump back.
- 750-780 Subroutine; set up a new deck as a very long string C\$.
- 790-830 Pull a card from the deck; shrink the deck since it's now gone.
- 840 Makes up a new deck if the old one is too small.
- 850-880 Assign card point values to the aces and face cards.
- 890-940 Subroutine; deal a card face up; sound depends on card value and position.
- 950-1000 Subroutine; deal a card face down; make a "clank" sound since you cannot see the card.
- 1010-1030 Subroutine; make a slurping sound when the dealer takes your money.
- 1040-1050 Subroutine; make a "bang" sound when anyone goes bust.

VARIABLE LIST FOR BLACKJACK

In order of appearance:

- SC is the first location in screen character memory.
- QA and QB are keyboard and line-print register locations in memory.
- P1 and P3 are memory locations for the pitches of voices #1 and #3.
- AD and WF are attack/decay and waveform memory locations for voice #1.
- VL is the memory location of volume control for all voices.
- U\$ is an input string for the player's wager.
- W is the numerical value of U\$.
- V is set equal to W for comparison later.
- L\$ is the first character of U\$ for a player quitting option.
- B is the player's bank in dollars, initially set to zero.
- LP is the integer position of the last card printed, left to right.
- DP is the dealing position vertically. DP=1 for the dealer and 2 for the player's cards.

CP is present card position for the card print subroutine.
VC = 1 to 10, the value of the last randomly selected card.
DH = the points in the dealer's hand at present.
DA = the value of the dealer's aces if any, 1 or 11.
DV = the points in the dealer's hand for comparison to DH later on.
AA = 0 or 10, and increment for low or high aces.
PH = the points in a player's hand.
PA is similar to DA for the player's aces.
D\$ is the two-character suit and rank of a card for the display.
X\$ = D\$ for the display subroutine.
PJ is a flag set to one if player has a blackjack.
DJ is a flag set to one if the dealer has a blackjack.
DD is set to one to flag a player has doubled down.
H\$ is a GET character to select hit, stand, or double option.
ZD is a flag indicating player cannot double down, hand too high.
DB = 1 is a flag indicating that the dealer has busted.
C\$ is the entire deck of cards, 104 characters with a full deck.
R = 1 to 103 to select a two-character card symbol from the deck C\$.
N\$ is a rank symbol from A to K for aces to kings from C\$.
Y\$ is the suit symbol next to N\$, $D\$ = N\$ + Y\$$.
T\$ is the portion of C\$ to the left of the card chosen.
C = the number of characters left in C\$.
K is an index counter for card printout subroutines.

10



By now, nearly everyone has twisted and hopelessly lost himself with one of those marvelously frustrating multi-colored cubes invented by Erno Rubik, a mathematician from Hungary.

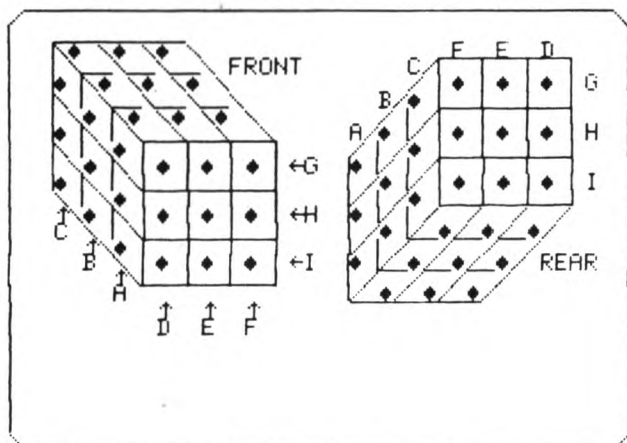
This program simulates the cube with a front and back display in which the colors are shown as diamond graphic symbols on the square faces of the cubelets. While the graphics do not rotate, the colors do, cubelet by cubelet for one complete twist each time. Sound effects accompany this, and there are ten levels of play rated 0 through 9. There are nine cubelets on each face of the magic cube. This game displays the front and back of the cube and asks for a level of difficulty, Z. The machine will randomly pick $Z+1$ sections of the cube and rotate these sections 90 degrees in a random direction, forward or reverse. Each section represents nine of the 27 cubelets making up the $3 \times 3 \times 3$ magic cube. The center cube is never visible and does not, in fact, exist inside the plastic toy cubes that are on sale. Instead, there is a very clever mechanism that allows free rotation of all the sections.

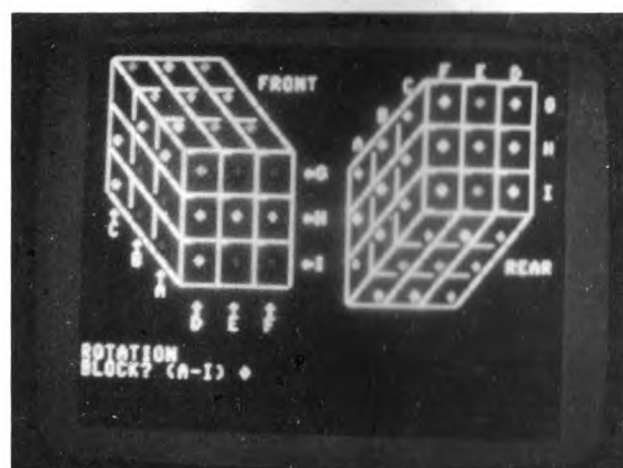
The machine next prompts the player to select which section to move. These are marked A through I on the screen. Having done this, he is further prompted for the direction to twist, 1 for forward and 2 for reverse. Arrows on the front display at the center of the screen indicate which direction is forward. The reverse display to the right of the screen enables the player to get a rear view of the cube.

The object of the game is to get the cube back into order with each side all of a single color. This is a lot to ask, especially at high levels of difficulty. To help out a player with a hopelessly scrambled cube, an algorithm was added at the end of the program to solve the puzzle. Just hit the S key and a rapid solution will follow, done by reversing all of the previous moves. The old moves are stored as characters in a long string that grows during ordinary play.

The difficult part for the reader is typing in all of those DATA statements near the end of the program. If errors occur, they will be obvious from the initial display when the diamond symbols are colored in. Any errors will leave some diamonds uncolored, or else colors will be mixed up on the sides of the cube. If you POKE color to a place on the screen where there is no corresponding character in character memory, then nothing will appear. In finding DATA errors, it is helpful to RUN the game at the zero difficulty level. You can move back the single random move easily and then exercise each section of the cube individually to see which one is the problem.

There are two types of cube sections: the end sections and the center sections. This accounts for the two different sizes of the arrays that hold the screen-color memory POKE numbers. Center sections have 12 colors to rotate, three at a time. End sections have this, too, but we need to rotate the entire side adjacent to the end section being rotated, or we have created an impossible object. Some cube-solving methods take advantage of this. One such method asks that you get all the corners in color agreement first and worry about the center section rotations later. Lots of luck.





LIST #10: MAGIC CUBE

```

1 REM*****
2 REM*MAGIC CUBE          COMMODORE 64 *
3 REM*BY LARRY HATCH      11.9 *
4 REM*MENLO PARK CALIF 94025 1983 *
5 REM*****
100 CLR:DIMA(20),C(20),D(20),F(20),I(20),G(20)
110 DIMB(12),E(12),H(12),R(20):QA=198:QB=214
120 SC=1024:CS=55296:POKE53269,0:POKE53280,9
130 VL=54296:P1=54273:P3=54287:AD=54277:WF=54276
140 PRINT"3":POKEQB,19:PRINT:PRINT"3MAGIC CUBE3"
150 FORI=0TO20:READA(I):NEXT:FORI=0TO20:READC(I)
160 NEXT:FORI=0TO20:READD(I):NEXT:FORI=0TO20
170 READF(I):NEXT:FORI=0TO20:READG(I):NEXTI
180 FORI=0TO20:READI(I):NEXT
190 FORI=0TO11:READB(I):NEXT:FORI=0TO11:READE(I)
200 NEXT:FORI=0TO11:READH(I):NEXT
210 POKE53281,0:GOSUB 830:M$="":REM*NEXT GAME
220 FORI=12TO20:POKEA(I),1:POKEC(I),2:POKEE(I),3
230 POKEF(I),4:POKEG(I),5:POKEI(I),6:NEXTI
240 POKEQB,20:PRINT:PRINT"SELECT LEVEL OF"
250 PRINT"DIFFICULTY (1-9) ◆":POKEQA,0:WAITQA,1
260 GETZ$:Z=VAL(Z$):PRINTZ$:POKEVL,11:POKEAD,30
270 FORU=0TO2:D=INT(2*RND(1)+1):B=INT(9*RND(2)+1)
280 POKEWF,0:M$=M$+CHR$(10*D+B+64)
290 ON B GOSUB490,520,540,570,600,620,650,680,700
300 IFB=2ORB=5ORB=8THENGOSUB790:POKEWF,0:GOTO320

```

```

310 GOSUB730:POKEWF,0
320 NEXTU:POKEQB,19:PRINT
330 PRINT"          ":REM 15 SPACES
340 PRINT"          ":REM + 19 SPACES
350 POKEWF,0:POKEQB,19:REM*ROTATION SELECT
360 PRINT:PRINT"ROTATION ":PRINT"BLOCK? (A-I) ◆III";
370 POKEQA,0:WAITQA,1:GETB$:PRINTB$
380 IFB$="*"THENPRINT"▲";GOTO210
390 IFB$="S" OR L>253 THEN1510:REM** SOLUTION
400 B=ASC(B$)-64:IFB<10RB>9THEN350
410 POKEQB,19:PRINT:PRINT"DIRECTION"
420 PRINT"1=FWD 2=RVS ◆III":POKEQA,0:WAITQA,1
430 GETD$:D=VAL(D$):PRINTD$:IFD<10RD>2THEN410
440 POKEQB,21:PRINT:PRINT"          ";
450 M$=M$+CHR$(10*D+B+64):L=LEN(M$)
460 ON B GOSUB490,520,540,570,600,620,650,680,700
470 POKEWF,0:IFB=20RB=50RB=8THENGOSUB790:GOTO350
480 GOSUB730:GOTO350
490 IFD=1THENFORI=0TO19:R(I)=A(I):NEXTI:RETURN
500 FORI=0TO11:R(I)=A(11-I):NEXTI:FORI=0TO7
510 R(I+12)=A(19-I):NEXTI:RETURN
520 IFD=1THEN FORI=0TO11:R(I)=B(I):NEXTI:RETURN
530 FORI=0TO11:R(I)=B(11-I):NEXTI:RETURN
540 IFD=1THEN FORI=0TO19:R(I)=C(I):NEXTI:RETURN
550 FORI=0TO11:R(I)=C(11-I):NEXTI:FORI=0TO7
560 R(I+12)=C(19-I):NEXTI:RETURN
570 IFD=1THEN FORI=0TO19:R(I)=D(I):NEXTI:RETURN
580 FORI=0TO11:R(I)=D(11-I):NEXTI:FORI=0TO7
590 R(I+12)=D(19-I):NEXTI:RETURN
600 IFD=1THEN FORI=0TO11:R(I)=E(I):NEXTI:RETURN
610 FORI=0TO11:R(I)=E(11-I):NEXTI:RETURN
620 IFD=1THEN FORI=0TO19:R(I)=F(I):NEXTI:RETURN
630 FORI=0TO11:R(I)=F(11-I):NEXTI:FORI=0TO7
640 R(I+12)=F(19-I):NEXTI:RETURN
650 IFD=1THEN FORI=0TO19:R(I)=G(I):NEXTI:RETURN
660 FORI=0TO11:R(I)=G(11-I):NEXTI:FORI=0TO7
670 R(I+12)=G(19-I):NEXTI:RETURN
680 IFD=1THEN FORI=0TO11:R(I)=H(I):NEXTI:RETURN
690 FORI=0TO11:R(I)=H(11-I):NEXTI:RETURN
700 IFD=1THEN FORI=0TO19:R(I)=I(I):NEXTI:RETURN
710 FORI=0TO11:R(I)=I(11-I):NEXTI:FORI=0TO7
720 R(I+12)=I(19-I):NEXTI:RETURN
730 FORJ=1TO3:Q=PEEK(R(11)):REM* ROTATION
740 FORI=10TO0STEP-1:POKEP3,5*J:POKEP1,Q+30+I*7
750 P=PEEK(R(I)):POKER(I+1),P:POKEWF,21:NEXTI
760 POKER(0),Q:NEXTJ:FORJ=1TO2:Q=PEEK(R(19))
770 FORI=18TO12STEP-1:P=PEEK(R(I)):POKER(I+1),P
780 POKEP1,9*P:NEXTI:POKE(R(12)),Q:NEXTJ:RETURN
790 FORJ=1TO3:Q=PEEK(R(11)):POKEWF,21:REM* CENTER
800 FORI=10TO0STEP-1:P=PEEK(R(I))
810 POKER(I+1),P:POKEP3,P+J:POKEP1,5*(I+P)
820 NEXTI:POKER(0),Q:NEXTJ:POKEWF,0:RETURN

```

```

830 PRINT " "
840 PRINT " "
850 PRINT " "
860 PRINT " "
870 PRINT " "
880 PRINT " "
890 PRINT " "
900 PRINT " "
910 PRINT " "
920 PRINT " "
930 PRINT " "
940 PRINT " "
950 PRINT " "
960 PRINT " "
970 PRINT " "
980 PRINT " "
990 PRINT " "
1000 PRINT " "
1010 PRINT " "
1020 TB=20:PRINT " ";
1030 PRINTTAB(TB)"
1040 PRINTTAB(TB)"
1050 PRINTTAB(TB)"
1060 PRINTTAB(TB)"
1070 PRINTTAB(TB)"
1080 PRINTTAB(TB)"
1090 PRINTTAB(TB)"
1100 PRINTTAB(TB)"
1110 PRINTTAB(TB)"
1120 PRINTTAB(TB)"
1130 PRINTTAB(TB)"
1140 PRINTTAB(TB)"
1150 PRINTTAB(TB)"
1160 PRINTTAB(TB)"
1170 PRINTTAB(TB)"
1180 PRINTTAB(TB)"
1190 PRINTTAB(TB)"
1200 REM* COLOR SCREEN LOCATIONS *
1210 DATA 55822,55702,55582,55503,55506,55509
1220 DATA 55638,55758,55878,55960,55963,55966
1230 DATA 55865,55745,55625,55628,55631,55751
1240 DATA 55871,55868,55748:REM* A(I) ARRAY
1250 DATA 55658,55538,55418,55339,55342,55345
1260 DATA 55482,55602,55722,55804,55807,55810
1270 DATA 55691,55571,55451,55448,55445,55565
1280 DATA 55685,55688,55568:REM* C(I) ARRAY
1290 DATA 55865,55745,55625,55503,55421,55339
1300 DATA 55451,55571,55691,55810,55888,55966
1310 DATA 55822,55702,55582,55500,55418,55538
1320 DATA 55658,55740,55620:REM* D(I) ARRAY
1330 DATA 55871,55751,55631,55509,55427,55345
1340 DATA 55445,55565,55685,55804,55882,55960

```

```

1350 DATA 55878,55758,55638,55560,55482,55602
1360 DATA 55722,55800,55680:REM* F(20) ARRAY
1370 DATA 55631,55628,55625,55582,55500,55418
1380 DATA 55451,55448,55445,55482,55560,55638
1390 DATA 55509,55506,55503,55421,55339,55342
1400 DATA 55345,55427,55424:REM* G(20) ARRAY
1410 DATA 55871,55868,55865,55822,55740,55658
1420 DATA 55691,55688,55685,55722,55800,55878
1430 DATA 55960,55963,55966,55888,55810,55807
1440 DATA 55804,55882,55885:REM* I(20) ARRAY
1450 DATA 55740,55620,55500,55421,55424,55427
1460 DATA 55560,55680,55800,55882,55885,55888:REM
1470 DATA 55868,55748,55628,55506,55424,55342
1480 DATA 55448,55568,55688,55807,55885,55963:REM
1490 DATA 55751,55748,55745,55702,55620,55538
1500 DATA 55571,55568,55565,55602,55680,55758:REM
1510 POKEQB,19:PRINT:PRINT"SOLUTION"
1520 PRINT"
1530 PRINT"
1540 L=LEN(M$):FORK=LT01 STEP-1:S$=MID$(M$,K,1)
1550 TW=ASC(S$)-64:D=INT(TW/10):B=TW-(10*D):D=3-D
1560 POKEQB,20:PRINT:PRINTD;CHR$(B+64)
1570 ONB GOSUB490,520,540,570,600,620,650,680,700
1580 IFB=20RB=50RB=8THENGOSUB790:GOTO1600
1590 GOSUB730:POKEWF,0:REM* VERY CLEVER!
1600 NEXTK:PRINT":":POKEQA,0:END

```

LINE ANALYSIS FOR MAGIC CUBE

- 100-120 Clear all variables; DIMension arrays; assign keyboard, screen line, character, and color POKE constants; turn sprites off; turn screen border brown.
- 130 Assigns the sound POKE constants.
- 140 Clears screen, PRINTs in white; PRINTs game title near screen bottom.
- 150-200 READ in color screen memory locations for the cubelets from DATA statements.
- 210 Colors screen field black; PRINTs cubes in perspective.
- 220-230 POKE colors into diamond graphics on cubelet faces.
- 240-260 Prompt and GET level of difficulty keystroke; set volume and attack/decay values.
- 270-320 Make Z+1 random twists of the magic cube.

- 320-340 PRINT over old prompts to erase them.
- 350-440 Player's next move, trying for a solution.
- 380 Allows player to start a new game using the asterisk.
- 390 Calls for the automatic machine solution if S-key is pressed or more than 126 moves are made.
- 400 Determines which section player wants to rotate.
- 430 Determines which way to rotate selected section of cube.
- 450 Stores the last move in order.
- 460 Jumps to the correct subroutine to rotate the selected section.
- 470 If an end section rotates, then rotate the entire adjacent cube face, too.
- 480 GOSUBs the actual rotation subroutine. Loops back for next move.
- 490-510 Assign R() array to values of A() array for rotation.
- 520-540 Assign R() array to values of B() array for rotation. The other arrays are fed into the R() array in an entirely analogous manner until line 720.
- 730-760 Perform actual rotation of colors for an end section wrap-around. Done in three steps.
- 760-780 Perform a two-step rotation of the colors on the adjacent cube side to an end section. Return from the end section rotation subroutine.
- 790-820 Perform actual rotation of colors for a center section wrap-around. All rotation subroutines have sound POKEs programmed in.
- 830-1190 PRINT in the main cube structure, front and rear.
- 1200-1500 DATA for screen color memory locations of diamond graphics on the magic cube.
- 1450-1460 DATA for the B center section color memory.
- 1470-1480 DATA for the E center section color memory.
- 1490-1500 DATA for the H center section color memory.
- 1510-End Subroutine to solve the puzzle.
- 1510-1530 Announce "solution"; blank out prompts.
- 1540-1560 Pull old moves, backwards, from M\$; character by character; calculate opposite moves; PRINT them to the screen.
- 1570-1600 Call appropriate subroutines to make all moves in reverse; pat player on the back; END.

VARIABLE LIST FOR MAGIC CUBE

In order of appearance:

A(I), C(I), D(I), F(I), G(I), and I(I) are arrays containing the 21 screen memory locations of the cube's faces for end sections. The first 12 are for the ring of four sides around this end section. The last nine are for the flat end face, which also rotates about its center.

B(I), E(I), and H(I) are arrays holding the 12 screen memories of the positions of the center sections, which rotate in groups of three.

R(I) temporarily stores the first group of arrays above for repositioning the numbers POKed into the screen memory locations.

QA, QB, SC, and CS should be very familiar by now.

I, J, K, and U are index counters for FOR-NEXT loops.

M\$ is string storage of each move made, including the original scrambling moves, one ASCII character per move.

Z\$ is a GET character for selecting level of difficulty.

Z is the value of the number string Z\$.

D = 1 or 2 for forward and reverse rotation of any section.

B = 1 to 9 = which section is to be rotated, A through I.

B\$ is A - I; player input for which section to rotate.

D\$ is "1" or "2"; player input for which way to rotate.

Q = temporary storage of a PEEKed cube screen location for rotation.

TB = 20 to TAB the printed display cube 20 places to the right.

L is the length of M\$ to keep it under 255 characters.

S\$ is each character from M\$ in turn to help unscramble the cube.

TW is a number indicating which twist of the cube S\$ represents so it can be untwisted.



MAGIC CIRCLES

Inspired by the Magic Cube game and toy, Magic Circles appears much simpler at first. Three intersecting circles having six main positions intersect one another at six points. After you select a level of difficulty from one to nine, the computer will randomly select circles and rotate them one way or the other. The result is a sort of barn dance for the 12 letters of the alphabet from A to L so that they can end up anyplace. It is up to the player to unscramble the mess and get the letters back into correct alphabetic order.

Even this simple layout gives us $N!$ (N -factorial) possible different arrangements of the letters. With $N = 12$ we have in fact over 479 million possible permutations or arrangements. Some of these cannot be reached with a low level of difficulty, of course, because they would require too much scrambling. The fact that the game is not as easy as it looks adds to the interest. Sound effects are included to reinforce your memory if you try to remember the moves and reverse them for a solution.

Any solution leading to the original alphabetic solution is valid, and there may be countless ways to find one besides reversing the scrambling that the computer does. Whenever the computer or the player rotates a circle, it is lit up in its appropriate color and filled in with stars of the same color. This helps you to see which one is going to rotate. If you should select the wrong circle (1 to 3) by accident, you can go back by hitting any letter key, other than S, and select the right one. The direction of rotation is selected by the 1 or 2 key.

If you want the computer to solve the puzzle and show an interesting sequence of moves, just press the S key. This jumps to a routine in the program that reverses all the previous moves. This is not the best solution, but it is the simplest one. It is a bit like stringing out thread in a forest so you can find your way out again if you get lost.

The solution routine is not a subroutine because we don't GOSUB to it or RETURN from it. In this context, a routine is no more lengthy or important than a subroutine. It is merely a matter of how we jump into and out of these program sections.

CIRCLES:			A	B
1				
2	3		C	D E
DIRECTIONS				
1	CLOCKWISE		F	G H I
2	COUNTER CW			
S	SOLUTION			
			J	K L
SELECT LEVEL OF DIFFICULTY (1-9)				




```

140 POKE53269,0:POKEAD,11:POKEVL,12
150 FORI=1TO6:READA(I),B(I),C(I):NEXTI
160 FORI=1TO6:READAS(I),BS(I),CS(I):NEXTI
170 FORI=1TO12:READD(I):NEXTI
180 PRINT:PRINT"INSTRUCTIONS? ♦";:POKEQA,0
190 WAITQA,1:GETZ$:PRINTZ$:IFZ$="Y"THENGOSUB570
200 FORI=1TO6:POKESC+A(I),90:POKESC+AS(I),42
210 POKESC+B(I),90:POKESC+BS(I),42:POKESC+C(I),90
220 POKESC+CS(I),42:POKECS+AS(I),7:POKECS+BS(I),3
230 POKECS+CS(I),5:POKECS+A(I),7:POKECS+B(I),3
240 POKECS+C(I),5:NEXTI:FORK=1TO12:POKESC+D(K),K
250 NEXTK:PRINT"30000 CIRCLES:0":PRINT"100"
260 PRINT"2 3000":PRINT"DIRECTIONS0"
270 PRINT"1 CLOCKWISE"
280 PRINT"2 COUNTER CW":PRINT"S SOLUTION"
290 M$="" :POKEQB,19:PRINT:PRINT"SELECT LEVEL OF"
300 PRINT"DIFFICULTY (1-9)":POKEQA,0:WAITQA,1
310 GETL$:LD=VAL(L$):IFLD<1THEN290
320 FORJ=1TOLD:RD=INT(2*RND(3)+1)
330 CN=INT(3*RND(1)+1):IFCN=COTHEN330
340 M$=M$+RIGHT$(STR$(CN),1)+RIGHT$(STR$(RD),1)
350 CO=CN:ONCNGOSUB910,930,950:GOSUB720:NEXTJ
360 FORI=1TO6:POKESC+AS(I),32:POKESC+BS(I),32
370 POKESC+CS(I),32:NEXTI:B=SC+800:FORI=BT0B+15
380 POKEI,32:POKEI+40,32:NEXTI:POKEQB,19:PRINT
390 PRINT"CIRCLE # ♦♦":POKEQA,0:WAITQA,1:GETZ$
400 PRINTZ$:CN=VAL(Z$):IFZ$="S"THEN490
410 IF CN<10RCN>3 THEN360
420 ON CN GOSUB910,930,950
430 POKEQB,20:PRINT:PRINT"DIRECTION ♦♦":POKEQA,0
440 WAITQA,1:GETZ$:PRINTZ$:RD=VAL(Z$)
450 IFZ$="S"THEN490
460 IFRD<1OR RD>2THEN360
470 M$=M$+RIGHT$(STR$(CN),1)+RIGHT$(STR$(RD),1)
480 GOSUB720:GOTO360
490 POKEQB,19:PRINT:PRINT"COMPUTER SOLUTION"
500 L=LEN(M$):FORK=1TOL STEP2:S$=MID$(M$,L-K,2)
510 CN=VAL(LEFT$(S$,1)):RD=VAL(RIGHT$(S$,1))
520 RD=RD-1:IFRD<1THENRD=2
530 ON CN GOSUB 910,930,950
540 GOSUB720:NEXTK:PRINT"COMPLETED"
550 PRINT:PRINT"NEXT GAME? ♦":POKEQA,0
560 PRINT" ":WAITQA,1:GOTO200
570 PRINT"♦ ** INSTRUCTIONS **":PRINT
580 PRINT"THE LEVEL OF DIFFICULTY YOU ENTER WILL"
590 PRINT"DETERMINE HOW MANY TIMES THE CIRCLES"
600 PRINT"ARE SCRAMBLED BY ROTATION. YOU TRY TO"
610 PRINT"SET THE LETTERS BACK IN ORDER BY THE"
620 PRINT"SAME SORTS OF ROTATIONS.":PRINT
630 PRINT"THE TOP CIRCLE IS #1; BELOW LEFT IS #2"
640 PRINT"AND BELOW RIGHT IS #3.":PRINT
650 PRINT"NEXT ENTER DIRECTION OF ROTATION (1-2)"
660 PRINT"WHERE 1 = CLOCKWISE AND 2 = C-CLOCKWISE"
670 PRINT:PRINT"AT ANY TIME YOU CAN START ANEW BY "

```

```

680 PRINT"USING THE 'S' KEY, THE COMPUTER WILL"
690 PRINT"ATTEMPT A SOLUTION OF ITS OWN."
700 PRINT:PRINT"PRESS ANY KEY TO PLAY.":POKEQA,0
710 WAITQA,1:POKEQA,0:PRINT"J":RETURN
720 N=2*CN+RD-2:REM ** ROTATIONS
730 ONNGOTO740,750,760,770,780,790
740 FORI=1TO6:R(I)=A(I):NEXTI:GOTO800
750 FORI=1TO6:R(I)=A(7-I):NEXTI:GOTO800
760 FORI=1TO6:R(I)=B(I):NEXTI:GOTO800
770 FORI=1TO6:R(I)=B(7-I):NEXTI:GOTO800
780 FORI=1TO6:R(I)=C(I):NEXTI:GOTO800
790 FORI=1TO6:R(I)=C(7-I):NEXTI:GOTO800
800 Q=PEEK(SC+R(6)):POKEP3,12+Q:FORI=5TO1STEP-1
810 POKEWF,19:P=PEEK(SC+R(I)):NT=10*I+20*CN
820 POKEP1,NT:REM** NOISE
830 POKE SC+R(I+1),P:NEXTI:POKE SC+R(1),Q
840 ON CN GOSUB 970,980,990:POKEWF,0:RETURN
850 DATA 107,304,310,113,310,316,316,513,519
860 DATA 513,710,716,507,704,710,304,501,507
870 DATA 70,267,273,195,392,398,435,632,638
880 DATA 550,747,753,425,622,628,185,382,388
890 DATA 107,113,304,310,316,501,507,513,519
900 DATA 704,710,716
910 FORI=1TO6:POKE SC+AS(I),42:POKE CS+AS(I),7
920 POKE CS+R(I),7:NEXTI:RETURN
930 FORI=1TO6:POKE SC+BS(I),42:POKE CS+BS(I),3
940 POKE CS+B(I),3:NEXTI:RETURN
950 FORI=1TO6:POKE SC+CS(I),42:POKE CS+CS(I),5
960 POKE CS+C(I),5:NEXTI:RETURN
970 FORI=1TO6:POKE SC+AS(I),32:NEXTI:RETURN
980 FORI=1TO6:POKE SC+BS(I),32:NEXTI:RETURN
990 FORI=1TO6:POKE SC+CS(I),32:NEXTI:RETURN

```

LINE ANALYSIS FOR MAGIC CIRCLES

100-120 Clear variables and screen; DIMension arrays; assign keyboard, screen line, character, and color POKE constants; screen field light grey; screen border dark grey.

130-140 Assign the sound POKE constants; turn sprites off; set attack/decay and volume for sound.

150-170 READ in arrays from DATA containing screen positions of the circle characters.

180-190 Prompt and branch for instructions.

200-240 Display the circles with diamond graphics in place of letter symbols.

250-280 PRINT mini-instructions showing circle placement and move keys.

290-300 Prompt for level of difficulty LD.

- 320-350 Scramble the circles LD times; record each rotation/direction in M\$.
- 360-370 Blank out the stars interpolated between all the letters.
- 370-380 POKE out all the old screen prompts.
- 390-410 Prompt for which circle to rotate.
- 420 Calls subroutine to highlight the chosen circle (1-3).
- 430-460 Prompt for which direction to rotate the chosen circle.
- 470 Records each two-keystroke move in M\$ (for move memory).
- 480 Calls the appropriate subroutine to actually rotate the chosen circle. Jumps back for the next player move.
- 490-540 Computer solution of the scrambled circles. The K-loop STEPs backwards through M\$ to GET the old moves.
- 510-520 Appropriate reverse motion moves are calculated.
- 530-540 Two subroutines called; highlight each circle and rotate the circle.
- 550-560 Prompt for a new game and go back to play it.
- 570-710 Subroutine to PRINT instructions, if called on line 190.
- 720-790 Start of real rotation subroutine; put the correct circle into the rotation array R(I) in correct order.
- 800-830 Rotate letter characters in the chosen circle with sound effects.
- 840 Turns the stars off: RETURNS from rotation subroutine.
- 850-880 DATA for all letter and star screen positions (0 to 999).
- 890-900 DATA for all letter positions only so we can POKE in diamond graphics at the start of the game.
- 910-920 Subroutine; highlight circle #1.
- 930-940 Subroutine; highlight circle #2.
- 950-960 Subroutine; highlight circle #3.
- 970-990 Three subroutines; turn off stars in each of the three circles.

VARIABLE LIST FOR MAGIC CIRCLES

In order of appearance:

A(I) I = 1 to 6 = the six screen positions of the first circle.

B(I) Likewise, the six screen positions of the second circle.

C(I) The screen positions of the third circle.

D(I) is all 12 circle number positions for first display.

QA, QB, SC, and CS are the usual screen, color, keyboard, etc., memory locations, as seen throughout these programs.

AD, P1, P3, VL, and WF are sound-generation memory locations as usual.

AS(I) are screen positions for the stars when circle 1 rotates.

BS(I) are second circle star screen positions.

CS(I) are third circle star screen positions.

Z\$ and L\$ are GET strings for processing player inputs.

M\$ is a long string containing all moves made so far. Each move is a two-character unit for circle and direction.

LD = 1 to 9 = level of difficulty.

I, J, and K are index counters in FOR-NEXT loops.

CN = 1 to 3 for which circle is to be rotated.

RD = 1 or 2 for clockwise and counterclockwise rotation.

CO is the last circle rotated for the random scrambling.

L is the length of M\$ in one-byte characters.

N = 1 to 6 to decide which array is to be rotated.

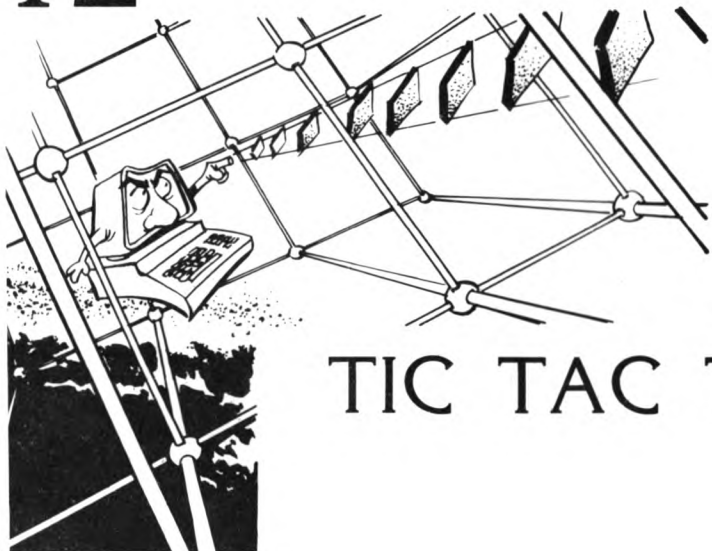
R(I) is the set of six screen character positions to be rotated.

Q is temporary storage of the last screen position in R(I). Q is also used to generate pitches for sound generation.

P and Q are used for temporary storage of first and last numbers in the R(I) array while rotating.

NT is another pitch for sound generation, placed in voice #1.

12



3-D TIC TAC TOE

This version of Tic Tac Toe uses a $3 \times 3 \times 3$ playing area shaped like a cube for three-dimensional play. The player has the machine for an opponent, and the winner is the first to place three men in any possible straight row. Since there are 49 such lanes counting all diagonals, the ability to think in three dimensions is encouraged.

In the first game in a RUN, the machine will take the first move. It then chooses the open center position on the middle level since this is the strongest square. With the advantage of the first move, and this strong position, the machine will be difficult if not impossible to defeat. There are few if any "cat's games" or stalemates in a playing area as compact as this; the author has never seen one.

Every second game, the player moves first, placing his reverse field X on the board by entering three coordinates at the keyboard. If he always chooses the center square (BY2), he will soon learn how to beat the computer consistently in these alternating contests. The first game is numbered zero.

A more interesting move is to pick a corner square when the player goes first. The machine will snap up the center square by default, but the player retains the advantage of the first move plus his own reasoned strategy. When the computer does not see a chance to win or a clear danger it will move at random.

During play, the machine will provide all the necessary prompts, allowing corrections for incorrect keystrokes if the user presses the N (negate) key. When the machine perceives a direct threat (two player's men in a row, the third place vacant), it will display "DEFENSE" along with the coordinates of the vital vacant square in that lane.

If it finds two of its own men, marked with diamond graphics, in a similarly opportune position, it will display "OFFENSE" and those coordinates. Consider the game over at that point. The machine will always choose the offensive move for a fast win, regardless of any threat present.

When the machine wins, it will flash its final diamond marker on and off and ring a bell several times to draw your attention to the deciding square. The display is less flamboyant when the machine loses.

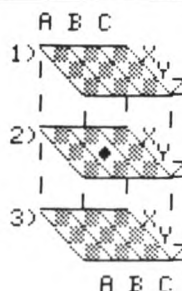
Player move coordinates are entered one at a time. First the "width" from left to right is entered as A, B, or C. Next is "depth," from the rear sloping forward in perspective, as X, Y, or Z. Finally, one of the three "levels" is chosen by number counting from the top down, like the flights in a Russian building.

Unlike crosstracks, this program uses an exhaustive search method to detect opportunities and threats, and it reacts to them directly. The squares are not given values for the machine to choose as in the "greed" algorithm. Finding no threats or opportunities, the machine simply picks a square at random! If the greed algorithm were used, the player who gets the first move would necessarily win unless he blundered. The machine would not blunder unless the program had some bug in it.

In this exhaustive search, every lane of three consecutive squares is examined in all three dimensions. First the machine tests the three levels separately, as though each level were a regular two-dimensional game. Next, the machine tests the vertical lanes, that is the same square on each of the three lanes descending. Next, the vertical diagonals are tested; for instance, A-Z-1, B-Z-2, and C-Z-3. These diagonals form 12 lanes descending from each of the four edges of the top level. Finally, the machine will test the extreme-corner to extreme-corner diagonals; there are four of these. All through these 49 lane tests, sound is generated by the ring-modulation technique. The index variables I, J, and K are used to POKE values into voice #3. The variable TS, which counts up the contents of each lane, is used to POKE numbers into voice #1. In ring modulation these two voices resonate off of each other for a sound effect that can only be called strange. You can fib a little bit and say that the machine thinks out loud.

3-DIMENSIONAL TIC-TAC-TOE

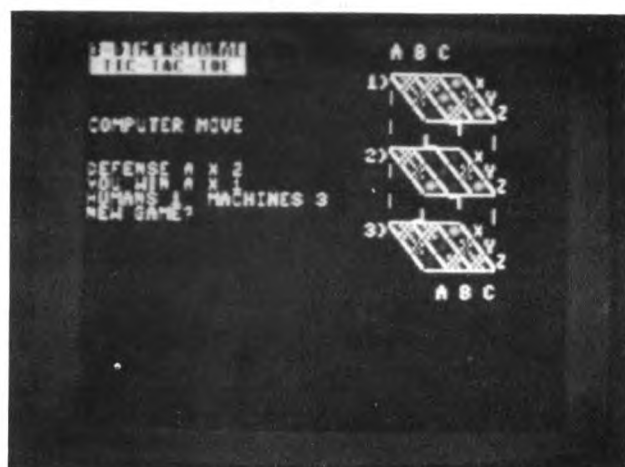
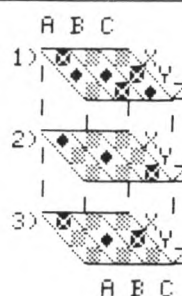
ENTER YOUR MOVE
WIDTH (ABC)A
DEPTH (XYZ)X
LEVEL (123)1

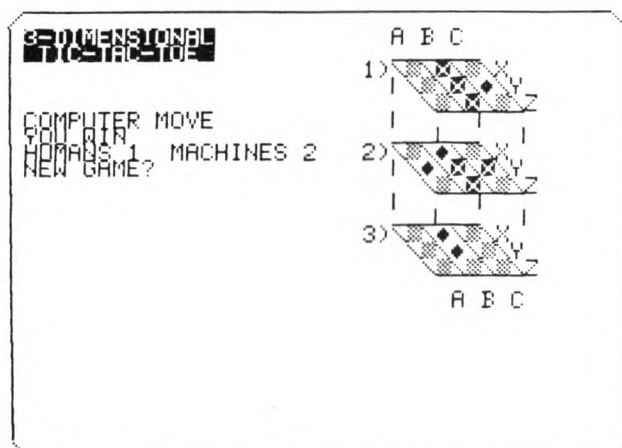


3-DIMENSIONAL TIC-TAC-TOE

COMPUTER MOVE

DEFENSE C Y 2
YOU WIN B Y 3
MACHINES 2
NEW GAME?





LIST #12: 3-D TIC TAC TOE

```

1 REM*****
2 REM*3D TIC-TAC-TOE    COMMODORE 64 *
3 REM*BY LARRY HATCH    V11.10 *
4 REM*MENLO PARK, CA 94025    1983 *
5 REM*****
100 CLR:POKE850,0:POKE851,0:POKE855,0:POKE53281,0
110 PRINT"3-D TAC TAC TOE":PRINT"3 TIC-TAC-TOE "
120 VL=54296:V1=54273:V3=54287:AD=54277:WF=54276
130 QA=198:QB=214:DIM B(2,2,2),P(2,2,2),C(2,2,2)
140 P(0,0,0)=1130:P(1,0,0)=1132:P(2,0,0)=1134
150 P(0,1,0)=1171:P(1,1,0)=1173:P(2,1,0)=1175
160 P(0,2,0)=1212:P(1,2,0)=1214:P(2,2,0)=1216
170 FORI=0TO2:FORJ=0TO2:FORK=0TO2
180 P(I,J,K)=P(I,J,0)+200*(K)
190 C(I,J,K)=P(I,J,K)+54272:NEXTK,J,I:FORI=0TO2
200 FORJ=0TO2:FORK=0TO2:B(I,J,K)=102:NEXTK,J,I
210 GOSUB670:G=PEEK(855):IFG/2=INT(G/2)THEN340
220 P1=0:P2=0:REM*** PLAYERS MOVE ***
230 GOSUB850:POKEQB,4:PRINT:PRINT"YOUR MOVE NEXT"
240 PRINT"WIDTH (ABC)":POKEQA,0:WAITQA,1:GETW$
250 D1=ASC(W$)-65:IFD1<0ORD1>2 THEN330
260 PRINTW$:PRINT"DEPTH (XYZ)":POKEQA,0:WAITQA,1
270 GETD$:D2=ASC(D$)-88:IFD2<0ORD2>3THEN330
280 PRINTD$:PRINT"LEVEL (123)":POKEQA,0:WAITQA,1
290 GETL$:D3=VAL(L$)-1:IFD3<0ORD3>2THEN330
300 PRINTL$
310 IFB(D1,D2,D3)<>102THENPRINT"OCCUPIED":GOTO330
320 B(D1,D2,D3)=81:GOSUB 870:GOTO340
330 GOSUB 850:GOTO220
340 BC=B(1,1,1):REM** MACHINE MOVES - 49 TESTS **
350 GOSUB850:POKEQB,4:PRINT:PRINT"COMPUTER MOVE

```

```

360 IF BC=102 THEN B(1,1,1)=90:GOSUB870:GOTO220
370 P1=0:P2=0:FORK=0TO2:FORJ=0TO2:TS=0:FORI=0TO2
380 K(I)=K:J(I)=J:I(I)=I:TS=TS+B(I,J,K):NEXTI
390 GOSUB930:NEXTJ:TS=0:FORI=0TO2:TS=0:FORJ=0TO2
400 K(J)=K:J(J)=J:I(J)=I:TS=TS+B(I,J,K):NEXTJ
410 GOSUB930:NEXTI:TS=0:J=0:FORI=0TO2:K(I)=K
420 J(I)=I:I(I)=I:TS=TS+B(I,I,K):NEXTI:GOSUB930
430 TS=0:FORI=0TO2:K(I)=K:J(I)=2-I:I(I)=I
440 TS=TS+B(I,2-I,K):NEXTI:GOSUB930:NEXTK
450 FORI=0TO2:FORJ=0TO2:TS=0:FORK=0TO2:I(K)=I
460 J(K)=J:K(K)=K:TS=TS+B(I,J,K):GOSUB930
470 NEXTK,J,I:FORI=0TO2:TS=0:FORJ=0TO2:I(J)=I
480 J(J)=J:K(J)=J:TS=TS+B(I,J,J):GOSUB930:NEXTJ
490 TS=0:FORJ=0TO2:I(J)=I:J(J)=2-J:K(J)=J
500 TS=TS+B(I,2-J,J):GOSUB930:NEXTJ,I:FORJ=0TO2
510 TS=0:FORI=0TO2:TS=TS+B(I,J,I):I(I)=I:J(I)=J
520 K(I)=I:GOSUB930:NEXTI:TS=0:FORI=0TO2:I(I)=2-I
530 J(I)=J:K(I)=I:TS=TS+B(2-I,J,I):GOSUB930:NEXTI
540 NEXTJ:TS=0:FORI=0TO2:I(I)=I:J(I)=I:K(I)=I
550 TS=TS+B(I,I,I):NEXTI:GOSUB930:TS=0:FORI=0TO2
560 I(I)=I:J(I)=2-I:K(I)=I:TS=TS+B(I,2-I,I):NEXTI
570 GOSUB930:TS=0:FORI=0TO2:I(I)=2-I:J(I)=2-I
580 K(I)=I:TS=TS+B(2-I,2-I,I):NEXTI:GOSUB930:TS=0
590 FORI=0TO2:I(I)=2-I:J(I)=I:K(I)=I
600 TS=TS+B(2-I,I,I):NEXTI:GOSUB930:REM END TESTS
610 IFP1=1 THEN GOSUB870:TS=270:GOTO990
620 IFP2=1 THENB(B1,B2,B3)=90:GOSUB870:GOTO220
630 A1=INT(3*RND(1)):A2=INT(3*RND(2))
640 A3=INT(3*RND(3)):A5=B(A1,A2,A3)
650 IFA5<102THEN630
660 B(A1,A2,A3)=90:GOSUB870:GOSUB850:GOTO220
670 PRINT"  " :TB=23:YW$="YOU WIN":IW$=" I WIN "
680 PRINTTAB(TB) "  A B C"
690 PRINTTAB(TB) "  _____"
700 PRINTTAB(TB) "1)\X\X\X\X"
710 PRINTTAB(TB) " | \Y\Y\Y\Y"
720 PRINTTAB(TB) "  \Z\Z\Z\Z"
730 PRINTTAB(TB) " |  _  _  _  "
740 PRINTTAB(TB) "  _  _  _  |"
750 PRINTTAB(TB) "2)\X\X\X\X"
760 PRINTTAB(TB) " | \Y\Y\Y\Y"
770 PRINTTAB(TB) "  \Z\Z\Z\Z"
780 PRINTTAB(TB) " |  _  _  _  "
790 PRINTTAB(TB) "  _  _  _  |"
800 PRINTTAB(TB) "3)\X\X\X\X"
810 PRINTTAB(TB) "  \Y\Y\Y\Y"
820 PRINTTAB(TB) "  \Z\Z\Z\Z"
830 PRINTTAB(TB) "  _____"
840 PRINTTAB(TB) "  A B C ":RETURN
850 FORI=0TO10:POKEQB,4+I:PRINT
860 PRINT" " :NEXTI:RETURN
870 FORI=0TO2:FORJ=0TO2:FORK=0TO2:REM X=0 UPDATE*
880 IFB(I,J,K)=81THEN POKEC(I,J,K),2
890 IFB(I,J,K)=90THEN POKEC(I,J,K),6

```

```

900 POKE P(I,J,K),B(I,J,K):NEXTK,J,I:POKEWF,0
910 POKEVL,15:POKEAD,12:POKEV3,31:POKEV1,133
920 POKEWF,21:RETURN
930 POKEWF,0:POKEV1,20+TS/3:POKEAD,12:POKEVL,12
940 POKEV3,I*3+J*5+K*7+10:POKEWF,21
950 IFTS=270ORTS=2820RTS=2640RTS=243THENGOSUB970
960 RETURN:REM* NO OBVIOUS THREATS
970 YS=PEEK(850):MS=PEEK(851)
980 IFTS=243 THENPRINT YW$:POKE850,YS+1:GOTO1120
990 IFTS=270 THENPRINT IW$:POKE851,MS+1:GOTO1120
1000 IFTS<>264THEN1060
1010 P2=1:FORQ=0TO2:IQ=I(Q):JQ=J(Q):KQ=K(Q)
1020 IFB(IQ,JQ,KQ)=102THENB1=IQ:B2=JQ:B3=KQ
1030 A$=CHR$(B1+65):B$=CHR$(B2+88):C$=STR$(B3+1)
1040 NEXTQ:POKEQB,7:PRINT
1050 PRINT"DEFENSE "A$ "B$;C$
1060 IF TS<>282THEN RETURN
1070 P1=1:FORQ=0TO2:IQ=I(Q):JQ=J(Q):KQ=K(Q)
1080 IFB(IQ,JQ,KQ)=102THENA1=IQ:A2=JQ:A3=KQ
1090 A$=CHR$(A1+65):B$=CHR$(A2+88):C$=STR$(A3+1)
1100 NEXTQ:POKEQB,8:PRINT
1110 PRINT"OFFENSE "A$ "B$;C$:RETURN
1120 YS=PEEK(850):MS=PEEK(851)
1130 PRINT"HUMANS"YS,"MACHINES"MS
1140 IF TS=243THEN1180
1150 Z=0:FORZ=0TO3:B(A1,A2,A3)=32:GOSUB870
1160 GOSUB1220:B(A1,A2,A3)=90:GOSUB870
1170 GOSUB1220:NEXTZ
1180 PRINT"NEW GAME? " :POKEQA,0
1190 WAITQA,1:CLR:PRINT"J":G=PEEK(855)+1
1200 GETZ$:IFZ$="N"THEN END
1210 POKE855,G:GOTO110
1220 FORZZ=0TO20:NEXTZZ:RETURN

```

LINE ANALYSIS FOR 3-D TIC TAC TOE

100-110 Clear variables; POKE zero scores into cassette buffer memory; color the screen field black; clear the screen; PRINT the title in reverse field and in white.

120-130 Assign all sound, keyboard, and line-print constants. DIMension the three-dimensional board, character, and color position arrays.

140-160 Assign screen character and color positions for the top level of the board.

170-190 Assign the screen position arrays for the center and bottom levels of the board.

190-200 Assign grey squares (102) for the entire board array. A grey square is vacant.

210-220 GOSUB the subroutine to PRINT in the play board; PEEK in the game count G; IF G is odd then the machine goes first at line 340.

230-300 Prompt and GET the player's three-keystroke move (ABC)+(XYZ)+(123); disallow irrelevant keystrokes.

310-320 Disallow choice of an occupied square on the board; claim a vacant square; mark it in on the board; jump to the machine's next move.

330 Erases old prompts and jumps back for a corrected player's move.

340-600 The machine examines the board to decide its next move.

350-440 The machine examines each lane horizontally, one level at a time.

440-470 The machine examines all vertical lanes straight up and down through the levels.

470-540 The machine examines vertical diagonals from side to side, from front to back, and vice versa. Twelve lanes are tested here.

540-600 The four extreme corner-to-corner lanes are examined here.

610-620 "Possible win" flags are tested, branching for appropriate countermeasures as needed.

630-650 Assuming no direct threat or possible immediate win, the machine randomly selects a square.

660 The square selected is claimed by the machine, printed in (GOSUB 870); old prompts are blanked out by the subroutine at line 850; jumps back for the player's next move. This concludes the main loop of the program.

670-840 Subroutine; send the cursor home; set endgame message strings; PRINT out the entire playboard. TABbed over to the right side of the screen; RETURN.

850-860 Subroutine; blank out all the old prompts and messages.

870-920 Subroutine; update the playboard by POKEing in characters and colors from the already updated arrays; RETURN.

930-960 Subroutine; make some sounds; test for obvious threats and opportunities; branch accordingly.

970-990 PEEK in previous scores from cassette memory; test for a winner; branch ahead if anyone won.

1000-1050 Test for player threat; prepare to block it.

1060-1110 Test for machine opportunity; prepare to act on it.

1120-1130 PEEK in and PRINT out the player and machine game scores.

1140 Branches ahead if the player won.

1150-1170 The machine won. Flash the winning move on and off; ring the bell each time.

1180-1210 Prompt for next game; PEEK in old game count; increment game count and POKE it right back; jump back for the next game.

1220 Delay loop for flashing and ringing if the machine wins.

VARIABLE LIST FOR 3-D TIC TAC TOE

In order of appearance:

V1 and V3 are the memory locations of voices #1 and #3 in sound generation. They are usually seen as P1 and P3. Since P1 is already used, V1 and V3 are used for pitches in this program.

VL, AD, and WF are the usual memory locations for volume, attack/decay, and waveform for sound.

B(I, J, K) where I, J, and K = 0 to 2 is the three-dimensional array of character POKEs for the three-layered game board.

P(I, J, K) is the game board array of screen memory POKE locations.

I, J, K, and Q are index counters for loops.

G is a game counter. If odd, the player moves first.

W\$ is the "width" coordinate of the player's move.

D1 = 1 to 3 corresponds to W\$ for the width coordinate.

D\$ is the "depth" coordinate of the player's move.

D2 = 1 to 3 corresponds to D\$ for the depth coordinate.

L\$ is the "length" coordinate of the player's move.

D3 = 1 to 3 corresponds to L\$ for the length coordinate.

P1 is a flag indicating an immediate possible machine win.

P2 is a flag indicating an immediate possible player win.

TS is a counter that steps through the various rows, columns, and diagonals to detect wins or possible wins.

I(n), J(n), and K(n), where n represents an index counter from zero to two, are temporary storage of the lane being examined. This is later used to find the empty spot in that lane.

A1, A2, A3 are randomly generated machine move coordinates.

A5 = PEEK B(A1, A2, A3) to test if the machine has chosen a free square.

TB = 23 is used as an argument for the TAB function in PRINT statements.

YS is the player's score in games.

MS is the machine's score in games.

IQ, JQ, and KQ transport values of I(), J(), and K() for convenience.

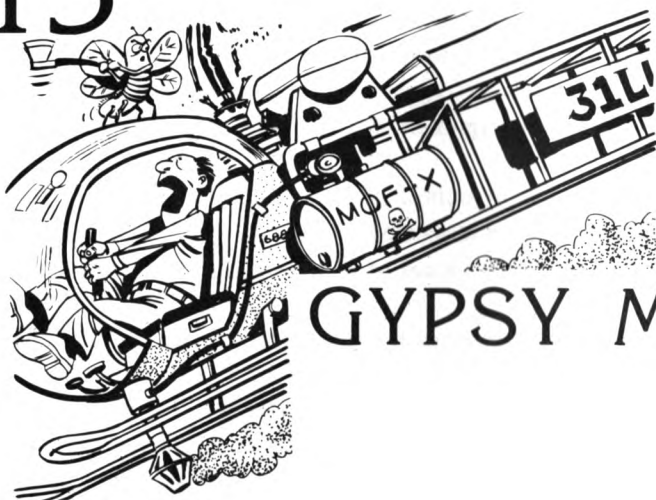
B1, B2, and B3 are used to generate announcements of the machine's moves.

A\$, B\$, and C\$ are themselves the announcement of machine moves.

A1, A2, and A3 are used also as coordinates of offensively planned machine moves.

Z and ZZ are used for delay loop index counters and to flash the winning move in place on the screen.

Z\$ is a simple GET string for player options.



GYPSY MOTH

Gypsy Moth is inspired by a recent infestation of gypsy moths in California and elsewhere. In this game, six swarms, each represented by a single graphic character, infest your screen. One swarm is active and mobile at a time until all of them are surrounded by pesticide-sprayed acreage or their own devastation. Areas where they have swarmed are marked by reverse-field X's representing the ruined crops.

The player controls a pesticide spray helicopter with the spray nozzle stuck on. He tries to surround the swarms before they can do too much damage. In so doing, however, he also ruins acres of land as a result of the poisonous spray. One square equals one acre. These sprayed fields are represented by grey squares. The object of the game is to minimize the total damage to the farmland and the crops grown there.

As each swarm is encircled by barriers of spray, or its own devastated areas, it becomes immobile and dies out for lack of food. Immediately, the next swarm comes out of dormancy, and the process is repeated until all six swarms have flown.

Chasing the active swarm is not the best strategy since the dormant ones are easier to encircle. Often, an active swarm will trap itself. When the last swarm is trapped, the game ends, and a score is computed based on the remaining untainted acres or spaces on the screen. These are left blank.

The designs that result from the combination of random and purposeful motion often have a curious appearance, something like modern art. Players who have a Commodore printer, which will print graphics characters, can run off copies or screendumps directly from the screen to paper.

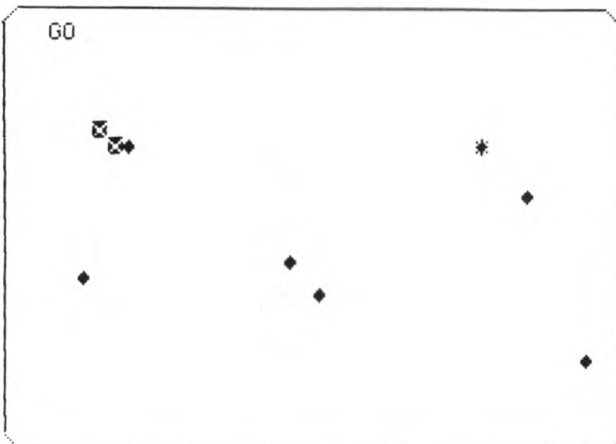
Simply choose the print option at the end of the game. A DEVICE NOT PRESENT error will occur if your machine does not see a printer attached when this option is chosen.

Note the OPEN 6, 4, 6 command on line 800. This tells the printer to space the lines together so the printout will look just like the screen. This works fine on my printer, an old Commodore 2022 attached through an interface and the IEEE-488 cable. If you have a printer, it is much more likely to be the Commodore 1525 type. If so, you may have to change lines 800 and 870. Please consult your printer manual. Six refers to channel six. Four is the device number of the printer. The last six is the "secondary address" on my printer. This is used for special commands to the 2022 type printer.

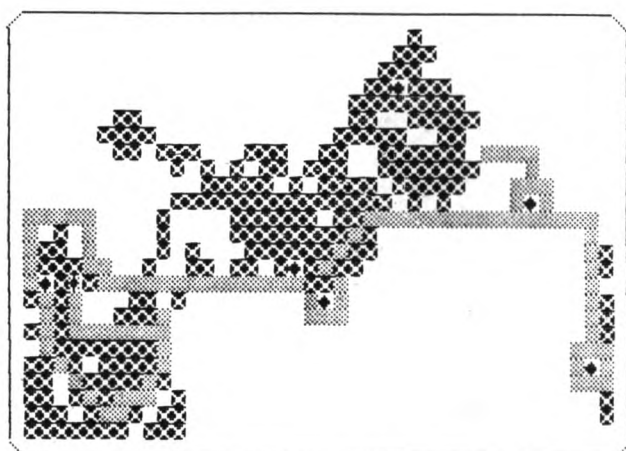
For a change of pace, or better objets d'art, you may wish to experiment with different values of FLY and JMP on line 320. FLY is the number of initial swarms. JMP is how many jumps the active swarm makes between player moves.

At any time, the player may terminate a game by pressing the S key. This jumps us to the scoring routine, followed by artwork printout options. This allows us to print our masterpieces when they are finished, even if the game itself is still in progress. The score shown is invalid in this case. It will be too high and should not count in any sort of competition.

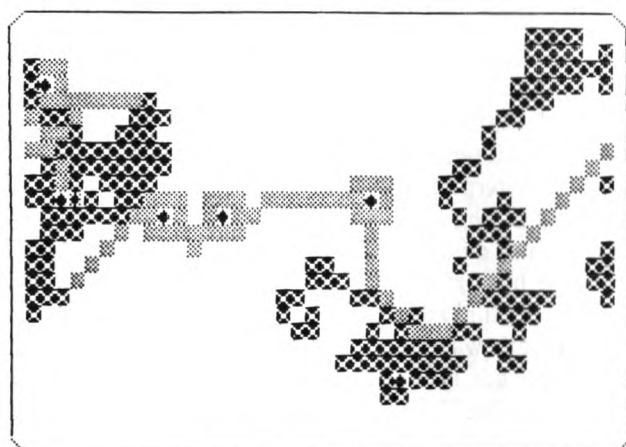
Naturally, we want to hear gypsy moth sounds. Each move of the helicopter sounds like a string plunked on a mandolin. This sound is heard via voice #2 using the sawtooth waveform. Voice #1 makes moth sounds (little fweeps) by using attack but no decay. Attack/decay for voice #1 is POKEd into memory location 54277. This program, like all the others, is in color even though the screen displays shown here may not make that obvious.



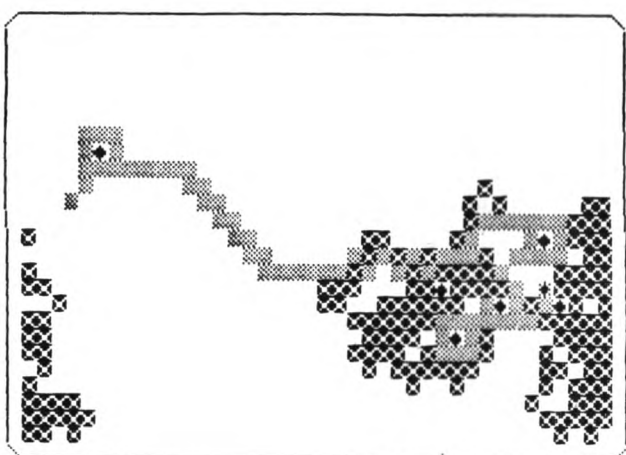
[Start of Game]



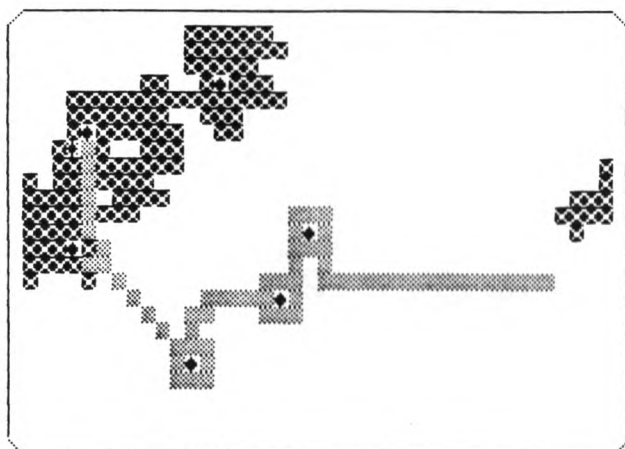
**Poodle with
Cocktails**



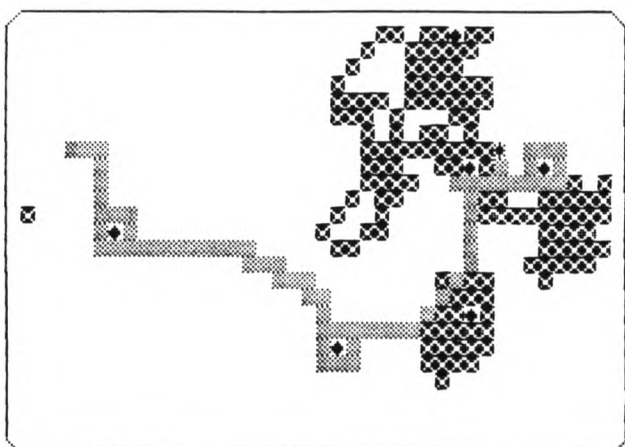
Stork and Child



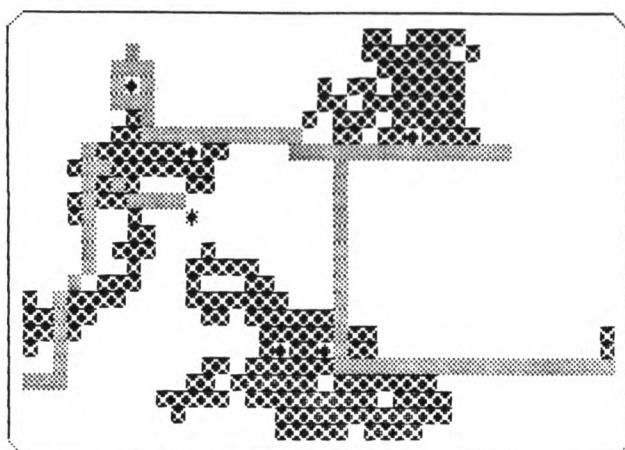
**Tyrannosaurus
Eating Giraffe**



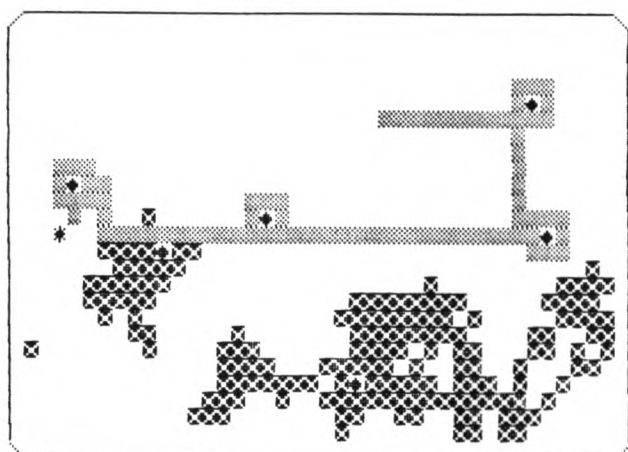
East Side Subway



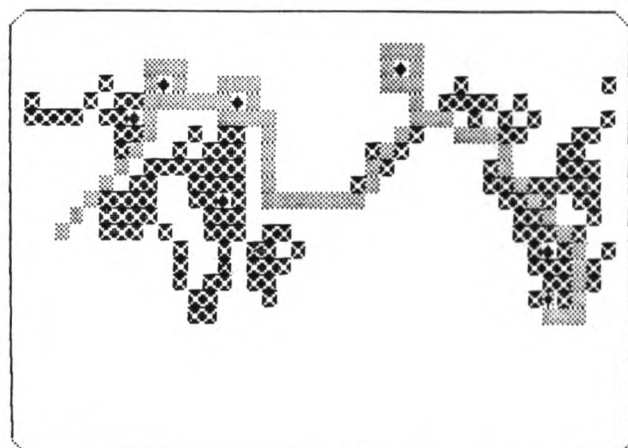
West Side Subway



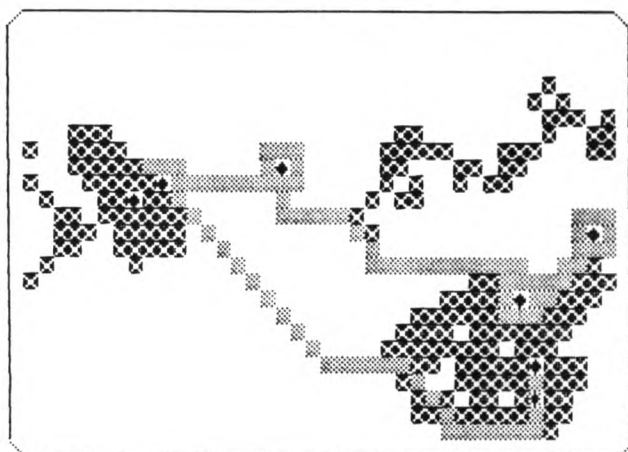
Man with Typewriter



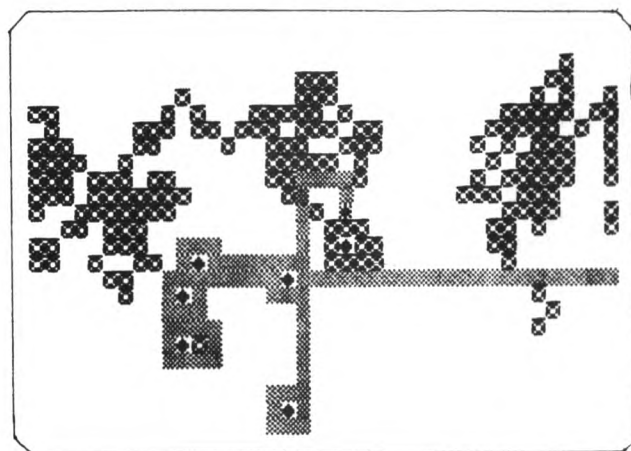
The Canoe



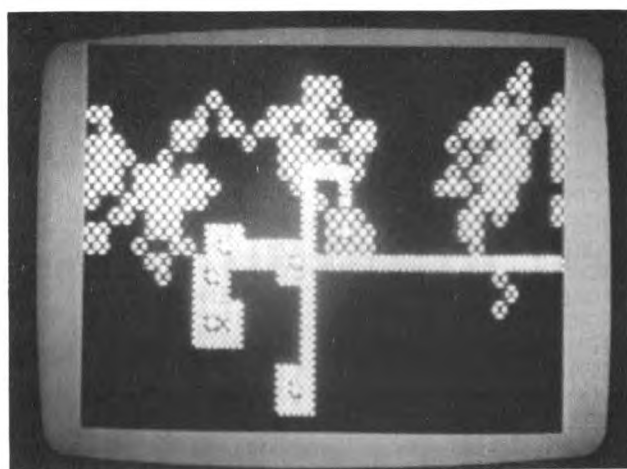
Golf Course



**The Western
Hemisphere Sideways**



4-Dimensional
Baseball



LIST #13: GYPSY MOTH

```

1 REM*****
2 REM*GYPSY MOTH      COMMODORE 64 *
3 REM*BY LARRY HATCH      10.21 *
4 REM*MENLO PARK, CA 94025      1983 *
5 REM*****
100 CLR:PRINT"GYPSY MOTHS!":SC=1024
110 DIM N(8),L(50),A(9),D(15,3):CS=55296
120 A(1)=39:A(2)=40:A(3)=41:A(4)=-1:A(5)=-39

```

```

130 A(6)=1:A(7)=-41:A(8)=-40:A(9)=-39:POKE53281,0
140 PF=100+INT(701*RN(1))
150 PRINT"THE GYPSY MOTHS ARE RUINING OUR CROPS!"
160 PRINT"YOU ARE IN A HELICOPTER SHOWN HERE-> *"
170 PRINT"SPRAYED ACRES ARE SHOWN THUSLY ■■■■."
180 PRINT"THE FERTILE MOTHS '♦' CANNOT ENTER THE"
190 PRINT"SPRAYED ACRES OR PREVIOUSLY INFESTED"
200 PRINT"ACRES 'X' THAT HAVE BEEN STRIPPED OF"
210 PRINT"FRUIT AND GROUND-SPRAYED."
220 PRINT"YOUR INSECTICIDE IS ALWAYS ON SO DONT"
230 PRINT"BREATHE.X"
240 PRINT"USE THE SHIFT AND CURSOR KEYS TO MOVE"
250 PRINT"HORIZONTALLY OR VERTICALLY."
260 PRINT"THE FLIES ARE TRICKY AND CAN JUMP OVER"
270 PRINT"DIAGONAL CORNERS."
280 PRINT"XPRESS 'P' FOR SCREEN PRINTOUT AFTER"
290 PRINT"THE GAME IS SCORED."
300 PRINT:PRINT"PRESS ANY KEY TO PLAY"
310 GETZ$:IFZ$=""THEN310
320 PRINT"J":FLY=6:JMP=2:REM TRY OTHER VALUES!
330 POKE53+PF,42:POKE53+PF,1:REM*NEXT GAME
340 P1=54273:P2=54280:VL=54296:POKEVL,10
350 W1=54276:W2=54283:POKE54277,64:POKE54284,7
360 FORI=1TOFLY:L(I)=100+INT(800*RN(3)+1)
370 POKE53+L(I),90:POKE53+L(I),3:NEXTI:I=1:FS=0
380 FORK=1TOJMP
390 IFI>FLYTHENK=JMP:NEXTK:GOTO650
400 FS=0:FORJ=1TO8:N(J)=L(I)+A(J)
410 IFPEEK(SC+N(J))=32THENFS=1
420 NEXTJ:IFFS=0THENI=I+1:GOTO390
430 POKEW1,0:R=INT(8*RN(3)+1):NX=SC+N(R)
440 NC=CS+N(R):GP=60+2*R+L(I)/70:POKEP1,GP
450 POKEW1,17:IFPEEK(NX)<>32THEN430
460 POKE53+L(I),214:POKE53+L(I),3
470 IFNX<SC+10THENL(I)=L(I)+40:GOTO400
480 IFNX>SC+999THENL(I)=L(I)-40:GOTO400
490 POKENX,90:POKENC,3:L(I)=N(R)
500 NEXTK:POKEW1,0:POKEW2,0:REM** SOUND OFF.
510 PRINT"GO":BN=BN+1:REM PLAYERS MOVE
520 GETD$:IFD$=""THEN520
530 PS=PF:PL=PF:IFD$="S"THEN650
540 DA=ASC(D$):IFDA=145THENNPD=-40:GOTO590
550 IFDA=17THENNPD=40:GOTO590
560 IFDA=157THENNPD=-1:GOTO590
570 IFDA=29THENNPD=1:GOTO590
580 GOTO520:REM* DISALLOW OTHER KEYSTROKES
590 PS=PS+PD:IFPS>999THENPS=PS-40
600 IFPS<10THENPS=PS+40
610 IFPEEK(SC+PS)=90THEN380
620 PF=PS:POKE53+PF,42:POKE 53+PL,102
630 POKEP2,GP/2:POKEW2,33:POKEW1,0
640 POKE53+PF,1:POKE53+PL,1:GOTO380
650 SC=1024:POKEW1,0:POKEW2,0:FORI=0TO15
660 FORJ=1TO3:D(I,J)=PEEK(SC+I+40*J):NEXTJ,I

```

```

670 D=0:POKEW1,0:PRINT"***SCORING***":FORR=0TO999
680 X=SC+R:NY=PEEK(X):IFNY<>32THEND=D+1
690 NEXTR:NS=1000-D
700 PRINT"SCORE:"NS:PRINT"PRESS  "
710 PRINT"P - PRINTOUT":PRINT"Y - NEW GAME"
720 GETZ$:IFZ$=""THEN720
730 PRINT"Z"P$ " ":IF Z$="P"THEN770
740 IFZ$="N"THEN POKEVL,0:END
750 IFZ$<>"Y"THEN700
760 PRINT"FRUIT FLY ":GOTO330
770 PRINT" " " ":REM MODERN ART PRINTOUT
780 FORI=0TO15:FORJ=1TO3:POKESC+I+40*J,D(I,J)
790 NEXTJ,I:SC=1024:OPEN1,4:PRINT#1:PRINT#1
800 OPEN 6,4,6:PRINT#6,CHR$(18)
810 FORI=0TO24:FORJ=0TO39:K=40*I+J
820 P$=" ":IFPEEK(SC+K)=214THENP$="X"
830 IFPEEK(SC+K)=90 THENP$="♦"
840 IFPEEK(SC+K)=42 THENP$="*"
850 IFPEEK(SC+K)=102THENP$="■"
860 PRINT#1,P$:NEXTJ:PRINT#1:NEXTI
870 PRINT#1:PRINT#1:CLOSE1,4:CLOSE6,4,6:GOTO760

```

LINE ANALYSIS FOR GYPSY MOTH

100-110 Clear variables; PRINT title in white; reverse-field; dimension variables; assign beginning character and color memory constants.

120-140 Assign values to the moth-jumping array A(n); turn the screen field black; assign PF a random starting position for the helicopter on the field.

150-300 PRINT instructions and prompt for a keystroke to continue.

310-320 Wait for the new keystroke; clear the screen; assign jump and fly variables.

330-350 POKE in the player's marker, an asterisk for the helicopter; assign sound variables; turn the volume up to 2/3 maximum; POKE in attack/decay values for voices #1 and #2.

360-370 The I-loop here assigns each swarm a random position on the screen and POKES it in, in color.

380-390 Here we index how many jumps each swarm gets and branch ahead if the last swarm has flown.

400-420 Examine all spaces around the Ith swarm, now active. If it is trapped, then increment I for the next swarm and jump back.

430-450 Randomly select which adjacent acre the swarm will infest. If occupied, then choose another one.

460-490 POKE the moth's old position with large X's to show the damage; make sure the new position is on the screen; adjust position if a swarm goes out of bounds; POKE the swarm into its new random position.

500 Ends the three-jump loop for the Ith swarm. Turns the sounds off.

510-530 Prompt and GET the next move for the player; increment the move counter; set his present and last position variables; jump ahead to the scoring routine if the player presses the S key.

540-580 Convert the player's keystroke to the position differential PD. Disallow irrelevant keystrokes.

590-620 Add PD to helicopter's present position; ensure that it is on-screen; don't let the helicopter jump directly onto the moth swarm, that would mess up the display. POKE the helicopter into his new position; POKE pesticide all over his old position.

630-640 Make a rather unlikely helicopter sound; jump back for the next swarm motions. This concludes the main loop of the program.

650-660 Save the upper lefthand corner of the display into the D(I, J) array for cold storage.

670-710 Count up the damaged acres D. Set score equal to 1000 minus D. PRINT this out in the upper left corner of the screen. Prompt for printout or the next game.

720-730 GET the next keystroke; jump ahead for artwork printout.

740-750 Allow options to stop the game or GOTO the next one.

760 Announces the next game. Now it's Fruitflies attacking! Jump back to spray fruitflies.

770-790 Put back the original corner of the screen art stored in the D(I, J) array.

790-800 OPEN channels to the printer; tell it to pack the lines together with no space between them.

810-850 Examine every square of the screen; set P\$ equal to the characters PEEKed from screen character memory, one by one.

860-870 PRINT#1, each character to the printer, all packed together (concatenated); send a dummy PRINT#1 out so the printer will line feed the paper after each line from the screen is complete. End the loop; give two extra line feeds; close all channels; go back for the next game.

VARIABLE LIST FOR GYPSY MOTH

In order of appearance:

$L(I)$ = 100 to 900 is the screen position of the I th gypsy moth.

$N(J)$ where $J = 1-8$ is the eight positions around the moth.

$A(J) = -41$ to $+41$ are increments representing screen distances to adjacent positions on the screen.

$D(15, 3)$ stores three lines of 15 characters so they can be replaced after screen prompts destroy them for the printout at the end.

$SC = 1024$ is the first screen character memory location.

$CS = 55296$ is the first screen color memory location.

PF is the position of the spraying flyer or helicopter.

$Z\$$ is a dummy GET string for a keystroke delay.

FL is how many moths appear at the beginning.

JM is how many jumps the active moth gets per player move.

FS is a flag equal to zero if a moth is stuck or surrounded.

$P1$ and $P2$ are memory locations for the pitches of voices #1 and #2.

$W1$ and $W2$ are memory locations for the waveforms of voices #1 and #2.

54277 and 54284 are memory locations of attack/decay for the two voices.

I , J , and K are index counters for FOR-NEXT loops.

R is a random number from 1 to 8.

NX is the moth's next prospective screen location.

BN is a player's move counter for sound generation.

$D\$$ is the player's move GET string.

D is the numerical equivalent of $D\$$ from 1 to 8.

PS is the prospective new position of the player's aircraft.

PL is set to player's old position to POKE in grey (sprayed) areas.

R is used again as an index counter during scoring.

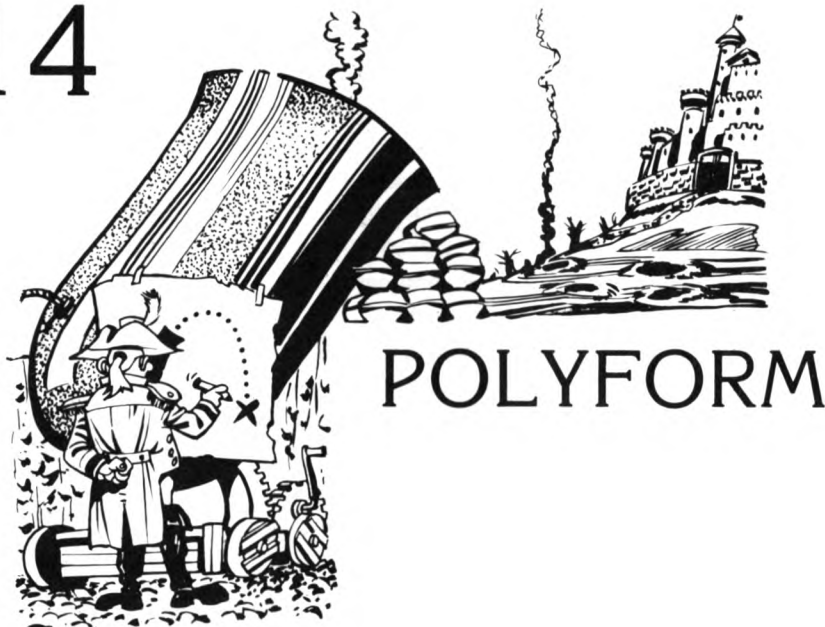
$X = SC + R$ to PEEK in each screen location in turn.

NY is the memory content of each screen location in turn.

D is a counter that increments for each destroyed acre of land.

NS is the new score equaling 1000 minus D .

$P\$$ is each character replaced into the area where the prompts appeared for the printout artwork.



Of all the kinds of mathematical functions or equations, polynomials are the best behaved. By this, I mean that they are regular, predictable, easy to manipulate in calculus and have other reassuring qualities. Graphs of polynomials are pleasing to the eye, and the functions that define these curves are more easily understandable than most.

Take a straight, level line for instance. The function $Y = 5$ would define such a line; Y remains equal to five no matter what value of X (from left to right) we choose. If we way more precisely that $Y = 1$ times X to the zero power, we are saying the same thing. Any number, even an unknown number, to the zeroth power equals one! This is the simplest polynomial, a polynomial of the zeroth degree, since the highest power of X is zero.

If our straight line ramps uphill as we draw our graph, then Y depends on X . One simple example is $Y = X$, or more properly, $Y = 1$ times X to the first power. In BASIC, we could state $Y = 1 * X^1$, but we usually don't. If we want a steeper slope, we could say $Y = 2 * X^1$. If we want to raise the whole curve without changing the slope, we could write $Y = 2 * X + 7$, or the like. Two points on a graph define a straight line. Get out your ruler and draw it; there is one straight line, and it is unique.

If we go to three points on a graph that are NOT on a straight line,

we define a circle, but a circle is not (alas) a polynomial function. Instead, we define a parabola, the trajectory of a cannonball on a flat earth with no wind resistance. Do three points define a unique parabola? They most certainly do. A parabola is a second-degree polynomial of the form $Y = aX^2 + bX + c$, where the coefficients a , b , and c can be any real numbers, positive or negative. Recall that we must write $A * X^2$, etc., where the asterisk (*) stands for multiplication. AX^2 is viewed as the single variable AX to the second power in BASIC.

So far, every group of N points on the graph defines a single (unique) polynomial curve and, therefore, a unique, simplest polynomial function of degree $N - 1$. For example, the three (X,Y) points $(-1,1)$, $(0,0)$, and $(1,1)$ define the second-degree polynomial $Y = X^2$, i.e., X squared. There is no other second-degree polynomial that will touch all three points. There are an infinite number of higher degree polynomials and other functions of nearly all descriptions that will cross those points, including the circle.

It is not all that obvious that one can generalize from these simple curves up to polynomials of any degree. Generalizations like this have broken down in the past. The proof has to do with the uniqueness of a solution to N polynomial equations of $(N - 1)$ th degree or less. Here we need N points on the graph for a unique polynomial function of the $(N - 1)$ th degree.

Given, say, eight points, we can find the unique (simplest) polynomial of seventh degree (or less!) by using this program. After some brief (and necessarily incomplete) instructions, the program asks how many points you want to enter in. For each point in turn, the machine asks the independent X value of the point and its dependent Y value. When all the N points are in, we first resort to the Lagrangian method of interpolating other values for the still unknown polynomial. We calculate $N + 1$ values of Y for X values of $0, 1, 2 \dots$ up to $X = N$. A subroutine at the end of the program takes care of this. Lagrange's excellent method is used again in the program for Fill the Curve (#19).

Having Y values for consecutive X values enables the rest of the method. The value of Y where $X = 0$ is our first coefficient immediately. We subtract this value from the Y values given by Lagrange's method. Next, the program takes the difference between adjacent Y values to obtain Y' values for the first numerical derivative. If these are all zero, then we have a first-degree polynomial.

Failing that, we subtract adjacent Y' values hoping for zero Y'' values, etc. When we finally zero-out, we have not only the degree of the polynomial but also a nice place to lay another zero. Even in infinitesimal calculus, the final derivative of any polynomial must be zero; we set all

values of Y''' (or whatever level of Y') to zero. Reversing our method, we build back up the pyramid of numbers by adding sums. This leads to the next coefficient of our polynomial function. This is repeated until we have all the coefficients and, therefore, the entire function.

This might seem like a nice way to chart common stocks or porkbelly futures, but it won't work there. The marketplace is the antithesis of the orderly polynomial functions. Perhaps some similar sort of program using the cyclical trigonometric functions would be of use in predicting stocks. A lot of work would have to be done just to filter out noise before waveforms of many different periods could be subtracted out for analysis.

Truncation error alone is a terrible problem with this method. That is why I have restricted the points that are entered to integer values. Theoretically, you could use fractional values, but you do so at your own risk.

For this program, it was necessary to build in truncation traps or lines that round off the calculations here and there. The final coefficients printed out are rounded to four decimal places. If you see $.1667 \cdot X^2$, just read "one-sixth times X squared." Please also recall that the fate of any polynomial is to sail off toward plus-or-minus infinity on either extremity of its graph. Small changes in points near the center of the graph will make great changes in the Y values toward these extremities.

ALL YOUR POINT PAIRS.

FROM THIS DATA, AN ALGORITHM WILL TRY
TO DETERMINE THE ORIGINAL FORMULA.

THE NUMBER OF POINTS MUST BE HIGHER
THAN THE DEGREE OF THE POLYNOMIAL.
FOR EXAMPLE, 4 POINTS WILL RESULT IN A
THIRD DEGREE POLYNOMIAL LIKE

$$X^3 - 2X^2 + 5X + 1$$

ALL X/Y VALUES MUST BE INTEGERS.

HOW MANY X/Y PAIRS DO WE PUT IN? 3

POINT# 1 X=? Y=? 1

POINT# 2 X=? Y=? 6

POINT# 3 X=? Y=? 11

CALCULATING.....
 COEFF# 0 = 5
 COEFF# 1 = -4
 COEFF# 2 = 2
 SOLUTION:

$$Y = 2X^2 - 4X + 5$$

$$X^3 - 2X^2 + 5X + 1$$

ALL X/Y VALUES MUST BE INTEGERS.

HOW MANY X/Y PAIRS DO WE PUT IN? 6

POINT# 1 X=2 Y=4

POINT# 2 X=3 Y=1

POINT# 3 X=2 Y=12

POINT# 4 X=3 Y=5

POINT# 5 X=2 Y=4

POINT# 6 X=3 Y=8

LAGRANGE'S METHOD

USES LAGRANGE'S INTERPOLATION METHOD
 TO FIND CONSECUTIVE VALUES OF THE MOST
 SIMPLE POLYNOMIAL FUNCTION CROSSING
 ALL YOUR POINT PAIRS.

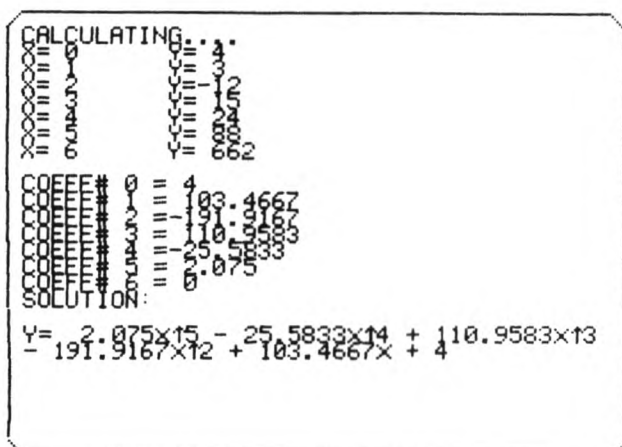
FROM THIS DATA, AN ALGORITHM WILL TRY
 TO DETERMINE THE ORIGINAL FORMULA.

THE NUMBER OF POINTS MUST BE HIGHER
 THAN THE DEGREE OF THE POLYNOMIAL.
 FOR EXAMPLE, 3 POINTS WILL RESULT IN A
 THIRD DEGREE POLYNOMIAL LIKE

$$X^3 - 2X^2 + 5X + 1$$

ALL X/Y VALUES MUST BE INTEGERS.

HOW MANY X/Y PAIRS DO WE PUT IN? 6



LIST #14: POLYFORM

```

1 REM*****
2 REM#POLYFORM      POLYNOMIAL FINDER *
3 REM#BY LARRY HATCH (C) 1983  11.15 *
4 REM#MENLO PARK CA      COMMODORE 64 *
5 REM*****
100 CLR:PRINT"POLYFORMULA FINDER"
110 POKE53269,0:POKE53280,9:POKE53281,0
120 PRINT"USES LAGRANGES INTERPOLATION METHOD "
130 PRINT"TO FIND CONSECUTIVE VALUES OF THE MOST"
140 PRINT"SIMPLE POLYNOMIAL FUNCTION CROSSING "
150 PRINT"ALL YOUR POINT PAIRS."
160 PRINT"FROM THIS DATA, AN ALGORITHM WILL TRY"
170 PRINT"TO DETERMINE THE ORIGINAL FORMULA."
180 PRINT"THE NUMBER OF POINTS MUST BE HIGHER"
190 PRINT"THAN THE DEGREE OF THE POLYNOMIAL."
200 PRINT"FOR EXAMPLE, 4 POINTS WILL RESULT IN A"
210 PRINT"THIRD DEGREE POLYNOMIAL LIKE"
220 PRINT"      X^3-2X^2+5X+1"
230 PRINT"ALL X/Y VALUES MUST BE INTEGERS."
240 INPUT"HOW MANY X/Y PAIRS DO WE PUT IN";N
250 DIM PX(N),PY(N),X(N),Y(N),CF(N),D(N),H(N,N)
260 FORI=1TON:PRINT"POINT#";I:INPUT"X=";PX(I)
270 PRINT"POINT#";I:INPUT"Y=";PY(I):PRINT:NEXTI
280 PRINT"CALCULATING..."
290 GOSUB540:Y(X)=S:PRINT"X="X,"Y="S:NEXTX:WN=0
300 CF(0)=Y(0):FORI=0TON:D(I)=Y(I):NEXTI:K=0
310 ZT=1:FORJ=1TON:D(J)=D(J)-CF(K)
320 IF ABS(D(J))>.0001THENZT=0
330 NEXTJ:IFZT=1THEN400
340 FORJ=1TON:D(J)=D(J)/J:H(0,J)=D(J):NEXTJ:A=0

```

```

350 FORJ=1TON-A-1:H(A+1,J)=H(A,J+1)-H(A,J):NEXTJ
360 IFACN=2THENA=A+1:GOTO350
370 FORB=AT00 STEP-1:H(B,0)=H(B,1)-H(B+1,0):NEXTB
380 K=K+1:CF(K)=H(0,0):IFK>N-1THENWN=1:GOTO400
390 GOTO310
400 PRINT:IFWN=1THENPRINT"POSSIBLE INPUT ERROR"
410 F$="Y= ":FORI=0TON
420 CF(I)=(INT((CF(I)+5E-5)*1E4))/1E4
430 PRINT"COEFF#"I"="CF(I):C$=STR$(CF(I))
440 L=LEN(C$):IFABS(CF(I))=1ANDI<>0THENC$=" "
450 C$(I)=RIGHT$(C$,L-1):NEXTI
460 PRINT"SOLUTION:" :FORI=N-1TO 0 STEP-1
470 IFCF(I)=0THEN530
480 S$=" + ":IFCF(I)<0THENS$=" - "
490 IFI=N-1 THEN S$=" "
500 F$=F$+S$+C$(I):E$=STR$(I):L=LEN(E$)
510 E$=RIGHT$(E$,L-1):IFI>0THENF$=F$+"X"
520 IFI>1THENF$=F$+"^"+E$
530 NEXTI:PRINT:PRINTF$:END
540 S=0:FORI=1TON:P=1:FORJ=1TON:IFJ=1THEN560
550 A=X-PX(J):B=PX(I)-PX(J):P=P*A/B
560 NEXTJ:S=S+PY(I)*P:NEXTI
570 SR=ABS(S-INT(S+.00001)):IFSR=0THEN590
580 IFSR<.00001THENS=INT(S+.00001)
590 RETURN

```

LINE ANALYSIS FOR POLYFORM

100-110 Clear variables; PRINT title in reversed field white; turn sprites off; color border brown, field dark grey.

120-220 PRINT instructions.

230-240 Prompt for how many points; DIMension variables accordingly.

250-260 GET X and Y coordinates of all N points.

270-280 Generate N + 1 values of X, starting from zero; call Lagrange subroutine to supply the corresponding Y values.

290-300 Use D(I) to subtract the first (easy) coefficient from all Y values.

310-320 Reset zero-test flag if any non-zero y" = D(I) value is found; branch ahead if none found.

330 Divides each y" by its position in the array! Replaces into the H(i,j) pyramid.

340 Subtracts adjacent values of y" in the H(i,j) array for the next level of numerical differentiation.

- 350 Narrows the pyramid as we descend to 2nd, 3rd . . . , etc., derivatives.
- 360 "Numerical anti-differentiation" straight up the side of the pyramid to establish the next coefficient of the polynomial.
- 370-380 Increment coefficient counter K; branch ahead if done; go back for next differentiation if not.
- 390 Announces possible error if WN (no-zero) flag is found set.
- 400-520 Set up the entire function display; F\$.
- 400-410 Round off and display each coefficient separately first.
- 420-440 Clean up the string representing each coefficient.
- 450-460 Announce a solution; skip over terms of zero value.
- 470-480 Ensure appropriate plus signs or no signs.
- 490-520 Put in multiplication signs, exponents with arrows, etc.; display the final F\$ polynomial function.
- 530-580 The Lagrange method subroutine.
- 530 Initiates a nested loop; prevents division by zero. Division by zero will occur here anyway if two original points have the same X value!
- 540-550 I still do not fully understand these two amazing lines.
- 560-570 Round off successive values of S, a "truncation trap"; close the loops and RETURN.

VARIABLE LIST FOR POLYFORM

In order of appearance:

- PX(n) is the X value of the nth point entered by the user.
- PY(n) is the Y value of the nth point entered by the user.
- CF(n) is the nth coefficient of the polynomial function to be found.
- D(n) is used to divide the y' , y'' , etc., values successively in the "numerical differentiation" routine.
- H(i,j) forms the inverted triangle of y' , y'' , y''' , etc., values as we differentiate downward.
- N is how many points are entered at the start. This number is used throughout the program.
- X indexes up to N to get $N + 1$ consecutive integer values of X, starting from zero for Lagrange's subroutine.

S is the Y value Lagrange returns for each value of X sent down to his subroutine.

WN is a flag set to zero if the differentiation routine zeroes out.

ZT is also a zeroed-out flag used to continue differentiation.

K, J, and B are index counters; J and B are in FOR-NEXT loops; K, which counts coefficients, is merely incremented.

F\$ builds up to be the final display of the function "Y = etc., etc."

C\$ is a string representation of each coefficient for clean display.

S\$ is a string representation of the sign of each term in F\$.

E\$ is a string representation of each exponent for clean display.

P is a "slope" between various points in the Lagrange subroutine.

B is used again as a set of differences between original X values in the Lagrange subroutine.

SR is used to round off values of S in the Lagrange subroutine.

15



There you are, minding your own business and writing traffic tickets, when the call comes in over the police radio. Some commie pinko is spray painting the streets red all over town! There is only one thing to do, give chase and paint the streets entirely blue! The villainous Red starts at the center of town and takes off randomly, leaving a wake of spray paint to mark his trail. The streets all start out nice and green. For every piece of green street that gets painted red, you lose a point. Scores can be positive or negative.

For every piece of green street that you paint blue, you gain a point. You gain two points for converting a red section to a blue one, but you lose two points when the red doubles back on your blue sections. At any time, if you are clever, you can circle around and catch the Red himself, but watch out. If you are one square out of step, he will run right over the top of your squad car and dump a bucket of paint on it.

Should your squad car become doused, it too becomes red. Then, all the streets you cover become red from the drippings. The only thing to do at that point is head for one of the three main intersections in the middle of town for a carwash. Crossing any of these four-way intersections converts you back to blue.

Meanwhile, there is this big gob of paint left where you got doused originally. If the Red steps on it, he will slip off the road and wander blindly

until he finds an intersection that he recognizes. This will occur randomly and affords you the opportunity to gather up a lot of points while the Red cannot subtract them back.

While your squad car is red, it cannot gain any points, certainly not from the mayor or the chief of police. Strangely, it is to your advantage to paint blue streets red on the way to the carwash! This gives you more streets to reconvert to blue without losing the points to turn them red first!

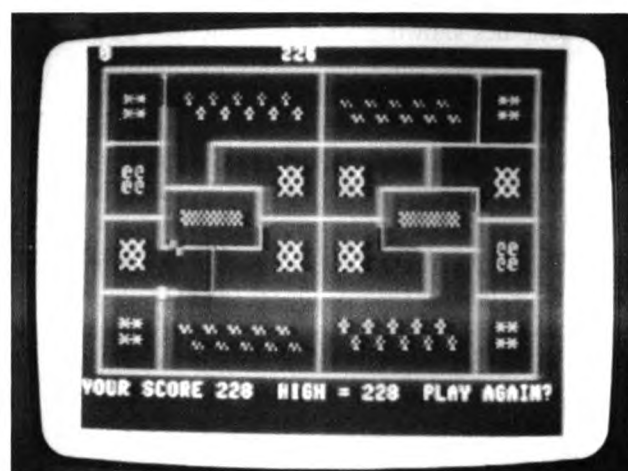
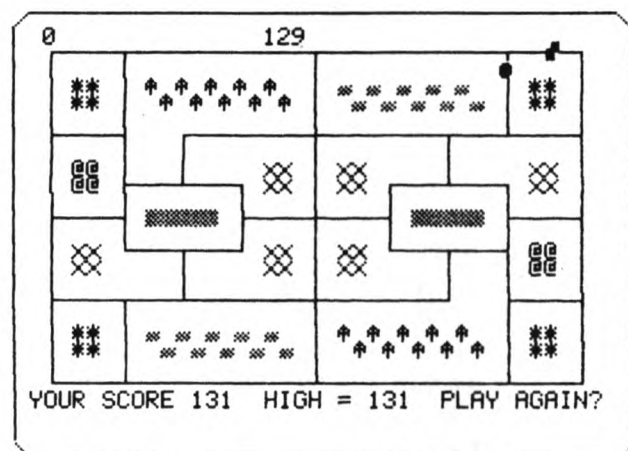
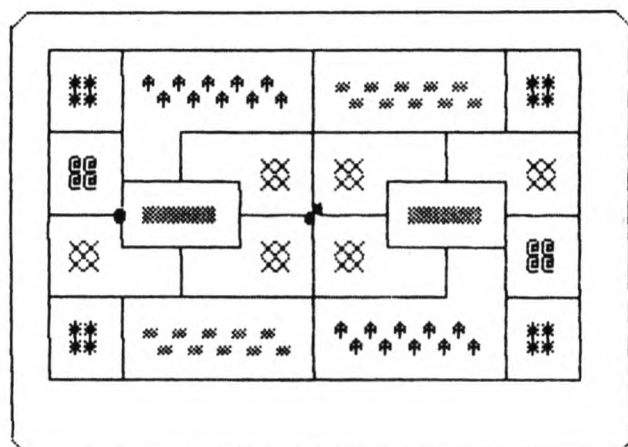
You have 500 moves. These count down in a small display at the upper-left corner of the screen. When these are gone, the game ends, unless you capture the Red first, which is risky. Scores of 150 and up are possible without capturing the maniac, but you will gain an instant 50-point credit if you do. This also ends the game. The Red is most easily captured head on. There is a 50-50 chance of a capture or of being doused on the spot, so think it over. This depends on who moves next when you are nose-to-nose. If you move first, that is a capture. The screen makes a nice display, and naturally, there is sound, but not sirens. You are rolling code-1.

Some care is needed in the screen print statements for the main display between lines 170 and 370. The horizontal lines used are the equivalent to screen POKE 64; that is the graphic you get with the shifted asterisk. Do NOT use the shifted C key, even if it looks the same! Likewise, the vertical line graphics are screen PEEK-POKE 93; which is the shifted minus sign. Don't use the shifted B key! All through the program, the computer is looking for certain numbers PEEKed from the screen to direct the men around.

Also, be careful not to use the shifted SPACE key when entering these lines. The shifted space is a screen PEEK of 96 instead of 32. If you change the scenery of the layout, keep everything at least one space away from the streets. What will happen (randomly), if you aren't careful with this, is the Red will wander off the streets where there is NO gob of paint.

The other graphics shown are the Commodore-key shifted A, S, L, and X keys for the corners; the Commodore-key shifted Q, W, E, and R keys for the T-junctions; and the regularly shifted plus sign key for the four-way crossings. There is nothing to prevent you from designing your own little town, but don't move the display up or down as a whole. The top line of the display is one line down from the top line of your screen; 21 lines of display in all.

The function keys F1, F3, F5, and F7 move the police car. F1 moves the car up. F3 moves it to the right. F5 moves it to the left. F7 moves it down.



LIST #15: REDS

```

1 REM*****
2 REM* REDS! FOR COMMODORE 64*
3 REM*BY LARRY HATCH 11.16*
4 REM*MENLO PARK, CA 94025 (C) 1983*
5 REM*****
100 CLR:DIMD(3):K(3)=40:K(4)=-40:K(5)=1:K(6)=-1
110 VL=54296:V1=54273:V3=54287:AD=54277:WF=54276
120 POKE53269,0:POKE53280,3:POKE53281,0
130 J1=112:J2=125:J3=109:J4=110:J5=107:J6=115
140 J7=113:J8=114:D(0)=-1:D(1)=1:D(2)=-40:D(3)=40
150 QA=198:QB=214:QC=197:SC=1024:CS=55296:T1=93
160 T2=64:J9=91:PRINT"  " :POKEVL,8:POKEAD,24
170 PRINT"
180 PRINT"
190 PRINT" **  # # # # #  * * * * * ** "
200 PRINT" **  # # # # #  * * * * * ** "
210 PRINT"
220 PRINT"
230 PRINT"
240 PRINT" @ @      X X      X X      X X      "
250 PRINT" @ @      X X      X X      X X      "
260 PRINT"
270 PRINT"
280 PRINT"
290 PRINT" X X      X X      X X      @ @      "
300 PRINT" X X      X X      X X      @ @      "
310 PRINT"
320 PRINT"
330 PRINT"
340 PRINT" **  * * * * *  # # # # # ** "
350 PRINT" **  * * * * *  # # # # # ** "
360 PRINT"
370 PRINT"
380 OL=SC+459:OP=91:D=D(4*RND(1)):LL=1064:UL=1902
390 CC=6:W=127:F=81:CL=2:S=0:M=500:P1=SC+446
400 C1=CS+446:POKEC1,CC:PP=115:POKEOL,W:POKEP1,F
410 NL=OL+D:IFNL<LL ORNL>ULTHENNL=SC+459
420 P=PEEK(NL):POKEOL,OP:W=382-W
430 OP=PEEK(NL):NC=NL+CS-SC
440 C=PEEK(NC):IFC<>2THENS=S-1:IFC=6THENS=S-1
450 POKEV1,140+50*COS(RM/7):POKEWF,20:POKEWF,21
460 PRINT"  "TAB(15)S"  "
470 POKENL,W:OL=NL:POKENC,CL:IFNL=P2THENCC=2:P=Q
480 IFPEEK(QC)>64THEN770
490 RM=RM+1:FORZ=1TO40:NEXTZ:REM* DELAY LOOP
500 IFP=T1ORP=T2THEN410
510 IFP<>J1THEN540
520 IFD=-40THEND=1:GOTO410
530 D=40:GOTO410
540 IFP<>J2THEN570
550 IFD=1THEND=-40:GOTO410
560 D=-1:GOTO410

```

```

570 IFP<>J3THEN600
580 IFD=40THEND=1:GOTO410
590 D=-40:GOTO410
600 IFP<>J4THEN630
610 IFD=1THEND=40:GOTO410
620 D=-1:GOTO410
630 IFP<>J5THEN660
640 R=INT(4*RND(1)):T=D(R):IFT=-DORT=-1THEN640
650 D=T:GOTO410
660 IFP<>J6THEN690
670 R=INT(4*RND(1)):T=D(R):IFT=-DORT=1 THEN670
680 D=T:GOTO410
690 IFP<>J7THEN720
700 R=INT(4*RND(1)):T=D(R):IFT=-DORT=40THEN700
710 D=T:GOTO410
720 IFP<>J8THEN750
730 R=INT(4*RND(1)):T=D(R):IFT=-DORT=-40THEN730
740 D=T:GOTO410
750 R=INT(4*RND(1)):T=D(R):IFT=-DTHEN750
760 D=T:GOTO410
770 MK=PEEK(QC):IFMK<30RMK>6THEN500
780 P2=P1+K(MK):Q=PEEK(P2):IFQ=32 ORP2<LLTHEN500
790 IFQ=127ORQ=255THENPOKEQA,0:S=S+50:GOTO860
800 C1=P2+CS-SC:CF=PEEK(C1):IFQ=91THENC=6
810 IFCC<>6THEN830:REM* NO POINTS GAINED
820 IFCF<>6THENS=S+1:IFCF=2THENS=S+1
830 POKEC1,CC:M=M-1:PRINT"  M"  ":IFM<1THEN870
840 POKEV3,9+23*ABS(SIN(M)):POKEP1,PP
850 PP=PEEK(P2):POKEP2,F:P1=P2:GOTO500
860 PRINT"BONUS: 50      "S
870 POKEQB,21:PRINT:IFS>HSTHENHS=S
880 PRINT"YOUR SCORE"S" HIGH ="HS" PLAY AGAIN?"
890 POKEQA,0:WAITQA,1:GETZF:GOTO160

```

LINE ANALYSIS FOR REDS

100-110 Clear variables; DIMension direction array; assign key direction array and sound memory location constants.

120 Turns sprites off; screen border to cyan; screen field black.

130-140 Assign screen PEEK recognition variables and direction array.

150-160 Assign keyboard, line-print, key-down, screen character and color, and other variables; clear the screen; PRINT in green; set volume attack and decay.

170-370 PRINT out the playing field.

380-400 Assign initial game variables including colors, positions, characters, and move counter; POKE the men in.

- 410 Keeps the Red on the screen.
- 420-430 PEEK at Red's next position; POKE back the street into his old position; toggle his graphic character; save an image of the street beneath his new position (OP again); calculate his new color screen memory location.
- 440 PEEKs into color memory of new position to decrement points where appropriate.
- 450-460 Make leftist noises and update score display.
- 470-480 POKE in Red's new character and color location on screen; test for a collision with the police; test for a key being pressed.
- 490 A delay loop in case the police don't move; keeps Red's speed more constant.
- 500 Jumps back for next Red move if no corners or junctions.
- 510-530 Test and turn an up-right corner.
- 540-560 Test and turn a down-left corner.
- 570-590 Test and turn a down-right corner.
- 600-620 Test and turn an up-left corner.
- 630-740 Test for T-junctions; randomly branch left or right; disallow backtracking.
- 750-760 Unconditionally turn (four-way intersection). Disallow backtracking. Jump back for next move.
- 770 PEEKs into QC for player's keystroke; disallows irrelevant keys.
- 780 PEEKs player's chosen position; disallows blank or off-street squares.
- 790 Tests for capture of the Red; bonus the score; branches.
- 800 Calculates new color position for blue; PEEKs into new character position; resets player to blue on four-way intersections.
- 810-820 Jump over scoring if the player is doused in red; otherwise, increment his score depending.
- 830 POKEs player's color into new position; decrements his moves-left; updates moves-left display; branches to end-game when out of moves.
- 840-850 Counterpoint leftwing sound; POKE blue into new position; put street back onto blue's old position; old player position becomes his new one for next time; go back to Red's move in progress.
- 860-890 The endgame. Starts on 870 if no capture made; increment high-score register for new highs; PRINT final results; wait for keystroke for the next game.

VARIABLE LIST FOR REDS

In order of appearance:

SC and CS are the screen character and color memory locations.

QA = 198 is a memory location counting keystrokes.

QB = 214 is a memory location controlling which line to PRINT on.

VL is the memory location controlling sound volume.

V1 in memory controls the pitch of voice #1.

V3 in memory controls the pitch of voice #3.

AD in memory controls the attack/decay rates of voice #1.

SR in memory controls the sustain/release of voice #1.

WF in memory controls the waveform of voice #1.

K(3-6) is the player's direction array.

J1-J8 are eight numbers PEEKed from the screen to test for street corners, etc.

D(0-3) are the direction array for the Red's random turns.

T1 and T2 are numbers PEEKed from the screen for straight street sections.

OL is Red's last screen character memory location.

OP is POKEed into OL location to put the street back.

LL and UL are the lower and upper limits of screen memory for the main display.

CC is the cop's color, usually 6 = blue.

W alternates between 127 and 255 to animate Red's display.

F = 81 is a ball character for the cop.

CL = 2 is the Red's color (i.e., red).

S is the player's score, always updated.

M is how many moves the player has left.

P1 is the player's present or old screen character memory location.

C1 is the player's color memory location.

PP stores the street graphic at Red's next screen character location.

NL is Red's next screen location in character memory.

P is the present contents of NL.

NC is the next color location for Red.

C is the contents of NC.

RM counts Red's moves.

P2 is the player's next screen character location.

Q is the contents of P2.

D is plus or minus 1 or 40, Red's present direction.

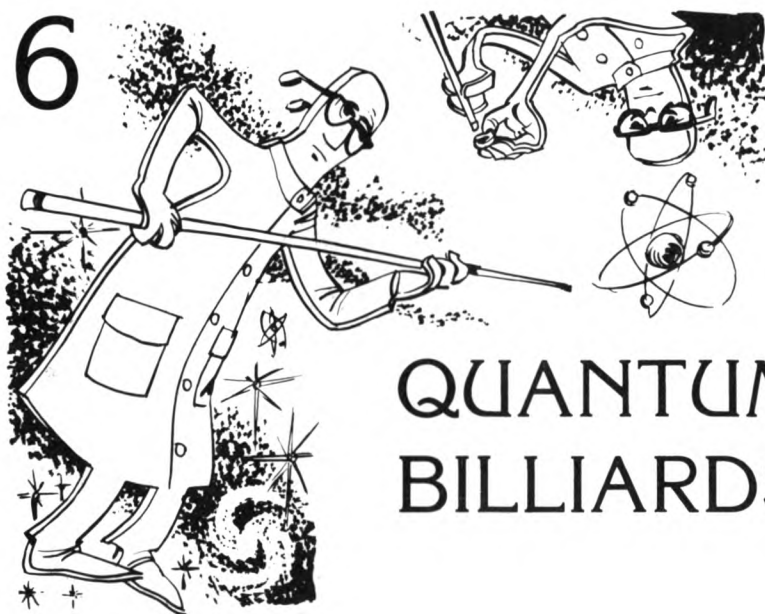
R is a random variable used to select D from D(R).

T is a temporary storage of D for testing.

MK is the key now down, PEEKed in from QC.

CF is the color found in the player's next position before moving there.

HS is the highest game score recorded to date.



QUANTUM BILLIARDS

Physicists have been shooting particles at each other for quite some time now. Why not use them in a sort of tiny billiards game? That was the original intent of this program. Problems arose immediately. The particles that came to be a part of this game don't behave like any we know of in nature. Staring at the computer screen, the solution became apparent. The particles used as billiard balls here are two dimensional. There is nothing more two dimensional than the image on a cathode ray tube. It has height and width but absolutely no depth at all.

Several magazine articles and at least one book have been written about "Flatland," a two-dimensional universe where the lack of depth necessitates an entirely different physics and chemistry. One article described all sorts of two-dimensional inventions like the 2-D water faucet and split-level apartments.

Classical Newtonian physics has been almost entirely worked out. For instance, in Flatland, gravity attracts as the inverse of distance rather than the inverse square of the distance. The "modern" physics of Flatland is open to conjecture, however, as this game may demonstrate. First of all, the third dimension of depth is as strange a concept to the little flatlanders as a fourth spatial dimension (not time) would be to us. Unlike these tiny people, we can easily see that their space is curved. If you doubt this, just take a short straight-edge and place it flat against your computer screen.

Being so small, the flat scientists can amuse themselves with Quantum Billiards. The game is played on a table which is 27 by 17 cells rectangular. These cells represent the only positions in two-space that two-dimensional subatomic particles can occupy! There is discontinuity of position so that a particle trajectory is really a series of discreet little hops from one cell to the next. This is ideal for the Commodore 64. We can use screen-POKE position to represent the cells.

There are no billiard cues or sticks. What could they make them out of? Instead the particles, or balls, are fired out of accelerators at the corner and side pockets. The side walls are non-absorbing barriers, possibly 2-D mirrors. The same pockets accept the incoming particles and store (conserve) the momentum for a very energy-efficient game.

All particle motion is diagonal since horizontal or vertical motion would result in endless loops. The particles strike and rebound from the walls at 45° angles as a result.

For any path that a particle can take, there is one entry and one exit point. Due to careful choice of dimensions 27 by 17, the eight possible paths criss-cross each other in a sort of tilted Scottish plaid pattern that covers every square on the board. All shots will bank or rebound 19 times before a particle is absorbed.

The game starts when the player chooses how many balls per game. If he chooses eight, for instance, then 16 "Billarons" appear. Eight will be solid blue balls representing the player's particles. Then eight hollow yellow circles appear representing the machine's particles. Both particles are of equal mass.

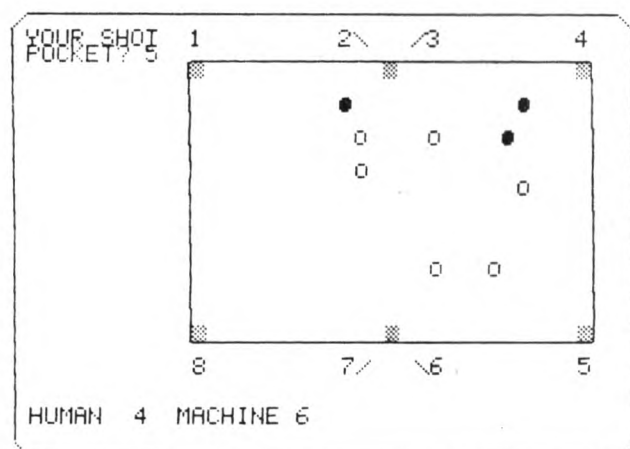
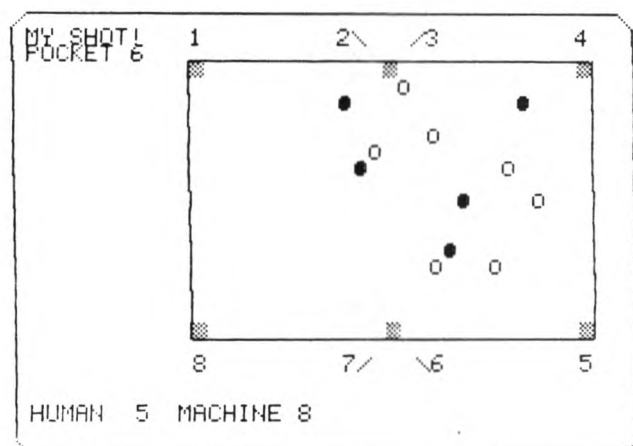
The player then chooses one of eight pockets to shoot a photon. The two side pockets each have two numbers because you can shoot in two directions from a side pocket.

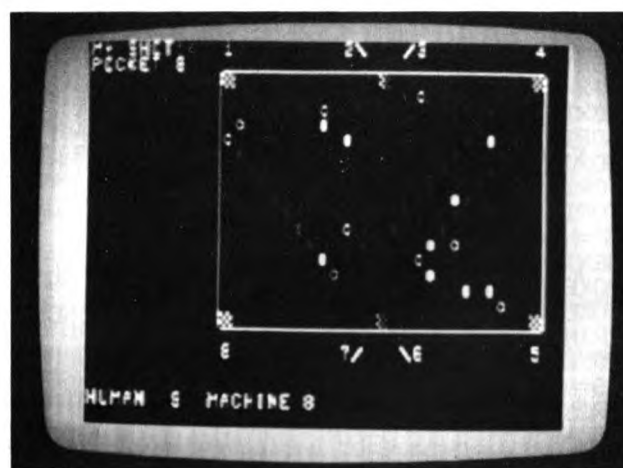
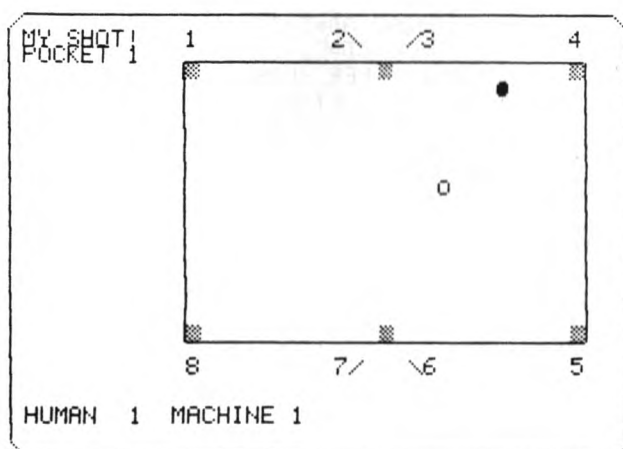
Next the pocket cyclotron revs up and fires a photon. Photon motion is slowed way down so the viewer can follow it. The photon banks around until it strikes a particle or disappears down another pocket. Being massless at rest, the photon vanishes as soon as it hits a billaron, yours or the machine's. This particle is then set into motion. It may sink into a pocket or knock yet another particle in. Several collisions usually occur, so it is hard to predict the result.

The machine shoots next. Turn rotates back and forth, regardless of a hit or a miss, according to Flatland billiard regulations. When the machine shoots, somewhere in its trajectory there is one of its own target particles. This particle may knock in one of the player's particles, but one of the machine particles will move. This is guaranteed by a mathematical algorithm that gives the machine its advantage.

The player's advantage lies in his ability to see the entire screen at once and plan out his shots in advance. He can see the geometry of the situation and plan an overall strategy.

Naturally, we want sound effects. Rather than mimic the sounds of an ordinary billiards table, and to keep the program at a reasonable length, I used the sounds of tiny two-dimensional particles and cyclotrons. We could never hear such noises ourselves. We are three-dimensional and much too large. Fortunately, the SID synthesizer chip in the Commodore 64 can imitate these sounds fairly well.





LIST #16: QUANTUM BILLIARDS

```

1 REM*****
2 REM*QUANTUM BILLIARDS COMMODORE 64*
3 REM* BY LARRY HATCH (C) 1983 *
4 REM* MENLO PARK, CA 94025 V 10.3 *
5 REM*****
100 CLR:PRINT"QUANTUM ":PRINT"BILLIARDS"
110 DIM LC(2),RC(2),LS(2),RS(2),PK(8),A(8),B(8)
120 DIM X(18),Y(18):POKE53281,0:VL=54296

```

```

130 P1=54273:P3=54287:AD=54277:SR=54278:WF=54276
140 SC=1024:CS=55296:CD=CS-SC:QA=198:QB=214
150 PRINT"*****BALLS PER PLAYER (1-9)? ♦♦♦";
160 POKEQA,0:WAITQA,1:GETBS$:BS=VAL(BS$):PRINTBS$
170 B2=2*BS:IFBS<10RBS>9THENPRINT"7":GOTO150
180 PRINT"♦♦♦ QUANTUM " :PRINT"♦BILLIARDS"
190 PRINT"♦TAB(11)"1      2\    /3      4"
200 PRINTTAB(11)"          "
210 PRINTTAB(10)"  █          █          █  "
220 FORI=2TO16:PRINTTAB(10)"  █SPC(27)"I " :NEXTI
230 PRINTTAB(10)"  █          █          █  "
240 PRINTTAB(11)"          "
250 PRINTTAB(11)"8      7/    \6      5"
260 SC=1024:ZP=SC+50:LC(1)=1:LC(2)=8:RC(1)=4
270 RC(2)=5:LS(1)=2:LS(2)=6:RS(1)=3:RS(2)=7
280 PK(1)=1115:PK(2)=1128:PK(3)=1128:PK(4)=1141
290 PK(5)=1781:PK(6)=1768:PK(7)=1768:PK(8)=1755
300 A(1)=1:A(2)=1:A(3)=-1:A(4)=-1:A(5)=-1:A(6)=-1
310 A(7)=1:A(8)=1:FORI=1TO4:B(I)=1:B(I+4)=-1:NEXT
320 PHT=42:HARD=81:SOFT=87:MB=BS:UB=BS:FORI=1TOB2
330 X=INT(27*RND(1)+1):Y=INT(17*RND(3)+1)
340 IF (Y=10RY=17)AND(X=10RX=14ORX=27)THEN330
350 FORJ=1TOI-1
360 IFX=X(J)ANDY=Y(J)THEN J=I-1:NEXTJ:GOTO 330
370 NEXTJ:X(I)=X:Y(I)=Y:NEXTI
380 FORI=1TO10:FORJ=1TO8:POKEPK(J),127:NEXTJ
390 FORJ=1TO8:POKEPK(J),255:NEXTJ,I
400 FORI=1TO8:POKEPK(I),102:NEXTI
410 FORI=1TOBS:HB=ZP+X(I)+40*Y(I):POKE HB,HARD
420 CP=HB+CD:POKECP,3:POKEVL,6:POKEVL,0
430 NEXTI:FORI=BS+1TOB2:SBALL=ZP+X(I)+40*Y(I)
440 POKE SBALL,SOFT:CP=SB+CD:POKECP,7
450 POKE VL,6:POKEVL,0:NEXTI:REM PLAYER SHOOT
460 PRINT"♦♦♦YOUR SHOT":PRINT"POCKET? ♦♦♦";
470 POKEQA,0:WAITQA,1:GETSH$:SH=VAL(SH$)
480 AS=ASC(SH$):IFSH$="7"ORAS=20ORAS=148THEN470
490 PRINTSH$:IFSH<10RSH>8THEN470
500 PRINT"          " :PRINT"          " :GOSUB790
510 FIND=0:REM* MACHINE SHOOT ***
520 PRINT"♦♦♦MY SHOT! " :PRINT"AIMING  "
530 FORI=1TO27:FORJ=1TO17:SEARCH=ZP+I+40*J
540 LOOK=PEEK(SEARCH)
550 IFLOOK=SOFT THEN FIND=SEARCH:I=27:J=17
560 NEXTJ,I:R=INT(2*RND(1)+1)
570 FIND=FIND-ZP:FY=INT(FIND/40)
580 FX=FIND-40*FY:S=INT(FX+FY+.0001)
590 D1=S/2:IFD1<>INT(D1)THEN620
600 CHARM=1:D2=D1/2:IFD2=INT(D2)THEN CHARM=2
610 GOTO670
620 CHARM=INT(2*RND(7)+3)
630 IFS=50RS=70RS=90RS=170RS=190RS=27THEN CHARM=3
640 IFS=290RS=370RS=390RS=41THENCHARM=3
650 IFS=30RS=110RS=130RS=210RS=23THEN CHARM=4
660 IFS=250RS=310RS=330RS=350RS=43THEN CHARM=4
670 R=INT(2*RND(1)+1)

```

```

680 IF CHARM=1 THEN SH=LC(R)
690 IF CHARM=2 THEN SH=RC(R)
700 IF CHARM=3 THEN SH=LS(R)
710 IF CHARM=4 THEN SH=RS(R)
720 PRINT "POCKET" SH: GOSUB 790: GOTO 460
730 PRINT "GAME OVER": IF UB<1 THEN PRINT "YOU WIN! "
740 POKENP, 102: IF MB<1 THEN PRINT "I WIN!! "
750 PRINT: PRINT "TRY AGAIN?": POKEQA, 0
760 GETZ$: IF Z$="" THEN 760
770 PRINT " ": PRINT " "
780 GOTO 180: REM ** PHOTON SHOT FOLLOWS BELOW **
790 PHT=42: HARD=81: SOFT=87: LAST=32: PC=1
800 SP=PK(SH): CP=SP+CD: OP=SP: POKEAD, 32: POKESR, 0
810 POKEVL, 10: FORI=1 TO 30: POKEWF, 0: POKEP1, 30+5*I
820 POKEWF, 33: POKESP, 127: POKECP, 7
830 POKESP, 255: NEXTI: POKESP, 102: POKECP, PC
840 A=A(SH): B=40*B(SH): PARTICLE=PHT
850 NP=OP+A+B: POKEVL, 9: IF PEEK(NP)<>32 THEN 920
860 CP=NP+CD: POKENP, PARTICLE: POKECP, PC: POKEAD, 48
870 POKEP1, 130+A*4-PA: POKEWF, 0: POKEWF, 17
880 POKE OP, LAST: LAST=32: OP=NP: IF NP=SP THEN 850
890 POKESP, 102: POKESP+CD, 1: POKE OP, 32
900 IF ZZ=1 THEN ZZ=0
910 GOTO 850: REM * PARTICLE MOTION LOOP **
920 IF PEEK(NP)<>102 THEN 1020: REM * COLLISION
930 POKEVL, 10: POKEAD, 32: POKEOP, 32: CP=NP+CD
940 FORI=1 TO 12: POKEWF, 0: POKENP, 127: POKECP, 7
950 FORK=1 TO 10: NEXTK: POKEP1, 30+10*K: POKECP, 3
960 POKEWF, 33: POKENP, 255: NEXTI
970 IF PARTICLE=HARD THEN UB=UB-1
980 POKE QB, 22: IF PARTICLE=SOFT THEN MB=MB-1
990 PRINT: PRINT "HUMAN "UB" MACHINE"MB" "
1000 IF UB<10 OR MB<1 THEN 730: REM * TO END OF GAME
1010 PRINT " ": POKENP, 102: RETURN
1020 P=PEEK(NP): IF P=99 OR P=100 THEN B=B*-1: GOTO 850
1030 IF P=101 OR P=103 THEN A=A*-1: GOTO 850
1040 IF P=81 THEN LAST=PARTICLE: PARTICLE=HARD: ZZ=1
1050 IF P=87 THEN LAST=PARTICLE: PARTICLE=SOFT: ZZ=1
1060 PC=7: IF PARTICLE=HARD THEN PC=3
1070 IF LAST=PHT THEN LAST=32
1080 POKEWF, 0: POKEP1, PA: POKEAD, 9: POKEP3, 15+2*A
1090 POKEVL, 12: POKEWF, 21: GOTO 860

```

LINE ANALYSIS FOR QUANTUM BILLIARDS

100-120 Clear variables; title in white reverse field; DIMension arrays; screen field in black.

120-140 Assign sound, screen, color, keyboard, and line-print PEEK-POKE constants for later use.

150-170 Prompt and GET player keystroke for number of balls; disallow numbers out of range.

- 180-250 Clear screen; retitle game; print in white; print out the playing table display.
- 260-270 Assign pocket numbers to all pocket shot trajectories.
- 280-290 Assign screen memory locations to each pocket.
- 300-310 Assign horizontal and vertical increments for shots out of each trajectory (pocket).
- 320-330 Assign graphics characters for photons and both types of balls.
- 330-370 Assign random X, Y table positions for all balls; disallow pockets and balls sharing the same position.
- 380-400 Rotate the cyclotron pockets; turn them all grey again.
- 410-420 POKE in all of the player's ball particles (blue).
- 430-450 POKE in all of the machine's particles (hollow yellow).
- 460-500 Player's shot; he selects a trajectory (pocket).
- 460-470 Prompt and GET player's next keystroke.
- 480-500 Disallow irrelevant keystrokes; blank old prompts; call the fire-shot subroutine.
- 510-780 Machine chooses a shot.
- 530-560 Machine scans entire table for one of its balls.
- 570-580 X, Y table coordinates of the ball found are calculated.
- 580-660 A complicated routine to determine which trajectory must pass through the ball found.
- 670-710 Correct trajectory determined; random selection of which end (pocket) of trajectory to shoot from.
- 720 Announces machine's shot; calls subroutine to shoot it; goes back for player's next shot.
- 730-780 Endgame; winner announced; prompt for next game, etc.; blank old prompts; jump back for another game.
- 790-1010 A lengthy subroutine to fire any shot. If a winning shot is fired, we jump out of the subroutine to line 730 above. On any collision, we jump to the routine from line 1020 to line 1090 and back.
- 790-830 Rotate the cyclotron pocket; make accelerator sounds.
- 840-850 The first particle is a photon.
- 850-910 Advance any particle; branch for collisions; make particle noises.
- 920 Tests for absorption by accelerator pocket; branches.
- 930-960 Rotate accelerator; make accelerator sounds; gobble the particle.

- 970-990 Decrement the appropriate ball counter; announce new counts.
- 1000-1010 Test for and announce any winners; end of fire-shot subroutine.
- 1020-1090 Collision routine, sometimes looped into by fire-shot routine.
- 1020-1030 If a wall is struck, bounce off at the opposite angle and loop right back.
- 1040-1050 The ball struck becomes the new ball in motion.
- 1060-1070 Assign the proper color for new ball in motion; annihilate any photons involved in a collision.
- 1080-1090 Make sounds of colliding particles! Loop back to fire-shot subroutine in progress.

VARIABLE LIST FOR QUANTUM BILLIARDS

In order of appearance:

- LC(1-2) = 1 or 8 for the number of these left-corner pockets.
- RC(1-2) = 4 or 5 for the number of these right-corner pockets.
- LS() and RS() are for the left and right side corner pocket numbers, depending on shooting angle there.
- PK(1-8) is the screen PEEK-POKE location of all pockets.
- A(I) is the initial Y-axis increment for a photon emitted from the Ith pocket.
- B(I) is the initial X-axis increment for a photon emitted from the Ith pocket. If B(I) is +1 then Y-axis motion is downward on the screen.
- X(I) is the X coordinate of the Ith ball's screen position.
- Y(I) is the Y coordinate of the Ith ball's screen position.
- P1, P3, AD, SR, and WF are sound-generation constants. These are described in detail elsewhere.
- BS\$ and BS are the number of balls per player asked for at first.
- I, J, and K are FOR-NEXT loop index counters.
- SC is the memory location of the first screen character position.
- CS is the memory location of the first screen color position.
- ZP is the screen memory location of the first position in the billiard table display.

PHT (PH) = 42 to POKE asterisks representing photons to the screen.
HARD (HA) = 81 to POKE the solid circles representing the player's particles.

SOFT (SO) = 87 to POKE the hollow circles representing the machine's particles (billiards).

MB and UB count down the remaining machine's and user's balls as they are eliminated.

X and Y are used to randomly give coordinates to all balls on the "break" at the beginning.

HB is the screen memory location of each hard ball in turn to POKE it onto the screen.

SB (SBALL) is the screen memory location of the soft (hollow) balls.
SH\$, SH = 1 to 8 is the player's choice of a pocket to shoot a photon from.

AS is the ASCII number for SH\$ to filter out bad keystrokes.

SEARCH (SE) is each memory location on the billiard table in turn.

LOOK (LO) is what is PEEKed from SEARCH.

FIND = SEARCH provided LOOK finds a machine (soft) ball.

FX and FY are the billiard table coordinates of the machine ball at location FIND.

S = FX + FY plus a tiny increment to prevent truncation down when they are INTegered to whole numbers.

D1 and D2 help decide which pocket the machine will shoot from.

CHARM = 1 to 4 to decide which type of pocket, left corner, right corner, side, etc., for the machine to shoot from.

R = 1 or 2 to decide which of the two CHARM-type pockets to shoot from randomly.

SH is the number of the pocket the machine has chosen.

LAST (LA) = 32 to blank out the photon's last position when it jumps.

SP is the screen location of the pocket PK(SH).

A and B are increments added to the coordinates of the particle in motion so it will move correctly.

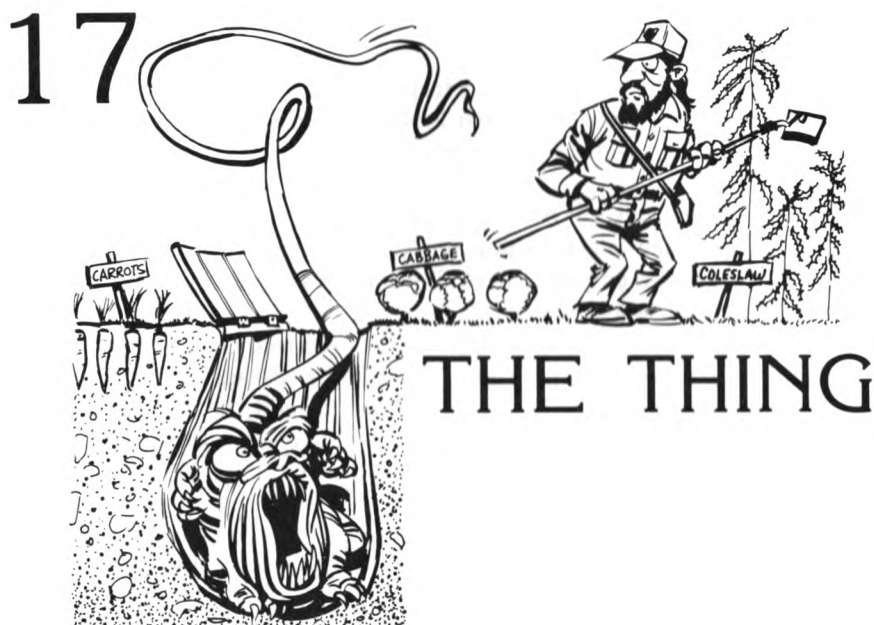
NP is the new location of the particle in motion.

OP is the old or previous position of the particle in motion.

PARTICLE (PA) is the type of particle; 42 for asterisk, etc.

P is the content of memory location NP.

ZZ is a flag to signal a collision of particles.



His spaceship out of control, the alien creature abandoned his craft in mid-flight and drifted to the ground. The ship struck a mountain top, forming Crater Lake. After searching down the coastline for rescue vessels, he finally curled up beneath the forest soil and went dormant to save energy. He slept for 650 years, until a gardener started farming herbs directly above him in the remote Humboldt Forest of northern California.

Naturally, this calls for eerie effects from the Commodore 64. Our program starts with the title and a movie-poster style blurb: "Awakened by the odd noises of gardening . . ." The sound of a strong wind through the trees is heard, using the noise waveform with an undulating pitch. Then a trapdoor opens, and the entire blurb is sucked in, character by character. This is followed by a theramin-like sound playing the descending notes of the "Thing Theme." When the blurb is consumed, a new screen of instructions appears, and play can proceed.

On the next keystroke, the screen clears, and a checkered character appears representing the player's man, the farmer. He can move up, down, left, and right by using the unshifted function keys F1, F3, F5, and F7. Every time the farmer moves, he leaves behind a seed, shown as a purple dot. After three moves, he is no longer safe; at any time the Thing can attack.

Randomly, the seeds will pop into ripe herbs. Herbs grow quickly and abundantly in Humboldt County. The ones in this game literally explode

into maturity. From time to time, a trapdoor will open and out comes the Thing—a sort of angular snake that looks like barbed wire. It makes random horizontal and vertical extensions to its length but always toward the farmer.

If the Thing runs out of length (20 vertebrae), it retracts into its hole and looks for a new place to pop up. If the Thing encounters a mature herb, shown as a green plant, it swallows it. A lump goes right down its neck; then it retracts as before.

The farmer soon discovers that he is the preferred meal, and he should hide behind some seeds. The Thing just hates seeds and will retract if it touches even one of them. If the farmer gets caught, he is “zorked” (in eight colors), swallowed, and the Thing retracts to end the game. The object of the game is to get the highest score possible by taking some amount of risk.

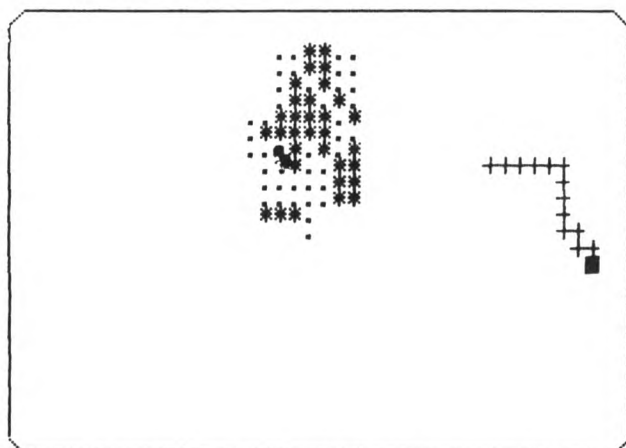
A good strategy is to build a solid, square plot of seeds without any openings inside. That trapdoor can appear in any open space. Don’t use up your allotted moves (500) by crossing needlessly over your own seeds. Instead, try to move only to sprouted herbs, ten points per plant harvested. As you move on, another seed is left to sprout later.

Don’t try to farm too much area; a six-by-six square is a good start. Too large an area is wasteful of moves and hard to defend. Remember that the Thing will eat up your profits if it can. It is often helpful to farm against the top of the screen. That way, you are protected on one side. Scores above 1500 are so-so for beginners. Scores above 2000 are OK to good. Scores above 3000 are very good. The theoretical limit of nearly 5000 is practically unattainable. There is no point penalty for getting zorked and swallowed, but it stops the farming prematurely. A count is kept of the highest game score since the program started for comparison.

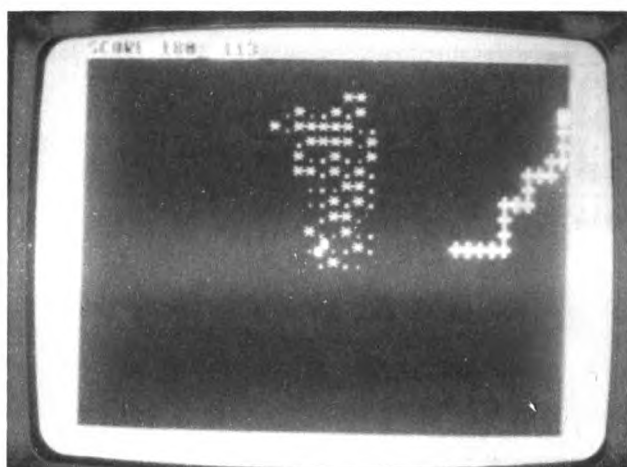
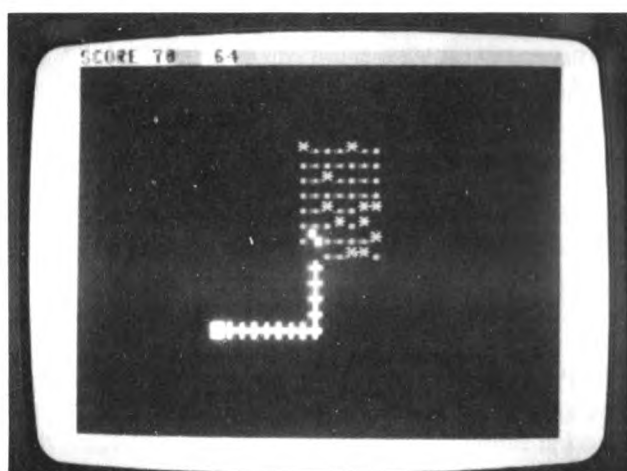
AWAKENED BY THE ODD NOISES OF GARDENING
THE QUIETEST DORMANT BEING IS SEARCHING
FOR HIS FIRST MEAL IN CENTURIES.....



Blurb



The Thing!



LIST #17: THE THING

```

1 REM*****
2 REM*THE THING!          COMMODORE 64*
3 REM*BY LARRY HATCH 11.17 (C) 1983 *
4 REM*****
100 CLR:PRINT" ";:SC=1024:CS=55296:POKE53269,0
110 QA=198:QB=214:P1=54273:P3=54287
120 AD=54277:SR=54278:WF=54276:VL=54296:P(0)=50
130 P(1)=47:P(2)=44:P(3)=35:FORI=54272TO54296
140 POKEI,0:NEXTI:POKE53281,0:POKESR,200
150 PRINT"THE THING IN THE FOREST GARDEN "
160 DIM TP%(21),M%(9),J%(1):FORZ=0TO200:NEXTZ
170 M%(0)=-40:M%(1)=1:M%(2)=-1:M%(3)=40
180 A$="AWAKENED BY THE ODD NOISES OF GARDENING"
190 B$=" THE ANCIENT DORMANT BEING IS SEARCHING"
200 C$=" FOR HIS FIRST MEAL IN CENTURIES....."
210 A$=A$+B$+C$:PRINT"*****"A$:POKEAD,255
220 POKEVL,10:POKEP1,128:POKEWF,129:FORI=0TO200
230 PT=20+ABS(10*COS(I/30)+11*SIN(I/11)+SIN(I/7))
240 POKEP1,PT:NEXTI:POKEWF,0:POKEAD,63:REM ↑ WIND
250 POKESC+435,160:POKECS+435,1:POKECS+395,1
260 L=LEN(A$):FORI=0TOL:R=PEEK(SC+355)
270 A$=" "+LEFT$(A$,L-1):PRINT"*****"A$
280 POKESC+395,R:POKEVL,6:POKEVL,0:NEXTI
290 PRINT" ";:FORZ=0TO39:PRINT" ";:NEXTZ:PRINT
300 PRINT"YOU ARE A GROWER CULTIVATING HERBS IN"
310 PRINT" THE FOREST WHERE THEY GROW FASTER."
320 PRINT"  YOU.  SEED  HERBS"
330 PRINT"  ++++  YOU MUST AVOID..."
340 PRINT"  ++  ←THE THING!"
350 PRINT"  USE FUNCTION KEYS TO MOVE:"
360 PRINT" F1=UP; F3=RIGHT; F5=LEFT; F7=DOWN"
370 PRINT"THE SEEDS LITERALLY EXPLODE INTO HERBS"
380 PRINT"YOU WANT THEM, SO DOES THE THING, BUT!"
390 PRINT"... HE WANTS YOU EVEN MORE.:POKEVL,9
400 FORH=0TO3:FORI=0TO3:POKEWF,17:POKEP1,P(I)
410 FORJ=0TO15:POKEP1-1,128+120*SIN(J):NEXTJ,I
420 POKEWF,0:NEXTH:REM* THERAMIN ↑↑↑
430 PRINT"***** PRESS ANY KEY TO PLAY":OC=32:M=0
440 POKEQA,0:WAITQA,1:TH=0:F=40:REM* NEXT GAME
450 PRINT" ";:FORZ=0TO39:PRINT" ";:NEXTZ:PRINT
460 YP=499:NP=499:YM=127:POKESC+YP,YM:POKECS+YP,1
470 Y1=INT(NP/F):X1=NP-F*Y1:GETM$:IFM$=""THEN560
480 A=ASC(M$):MV=A-133:OC=46:IFMV<0ORMV>3THEN500
490 NP=YP+M%(MV):M=M+1:PRINT" ":REM YOUR POSITION
500 YM=382-YM:POKESC+YP,OC:POKECS+YP,4
510 PRINT"TAB(10)M:IFNP<F ORNP>999THENNP=YP
520 IFM=>500THEN940:REM- 500 MOVE LIMIT
530 OC=PEEK(SC+NP):POKESC+NP,YM:POKECS+NP,1:YP=NP
540 IFOC=42THENS=S+10:PRINT"SCORE"S:OC=32
550 GOTO470:REM* RANDOM SPROUTING BELOW
560 FORI=0TO1:Q=F+INT(960*RND(1)):B=PEEK(SC+Q)
570 IFB=46THENGOSUB810:POKESC+Q,42:POKECS+Q,5
580 NEXTI:IFM<3THEN470

```

```

590 IFTH=1THEN660:← THING IS ACTIVE: JUMP AHEAD
600 R=INT(10*RND(1)): IFR=7THENTH=1:N=0
610 R%=3*RND(1)+1: IFTH<>1THENK=0:GOTO470
620 R=INT(25+50*RND(1)):P(R%)=R
630 K=0:TR=INT(960*RND(1)+F):TP%(0)=TR
640 IFPEEK(SC+TR)<>32THEN630
650 POKESC+TR,160:POKECS+TR,1:GOTO470
660 Y2=INT(TR/F):X2=TR-F*Y2:J%(0)=SGN(X1-X2)
670 J%(1)=F*SGN(Y1-Y2):R=INT(2*RND(1))
680 IFJ%(0)=0ANDJ%(1)=0THEN710
690 IFJ%(R)=0THENR=1-R:GOTO690
700 TR=TR+J%(R):K=K+1:TP%(K)=TR:V3=TR/8
710 IFK>20THENK=20:GOSUB790:GOTO470
720 F=PEEK(SC+TR):IFP=32ORP=91ORP=160 THEN760
730 IFP=46THEN TP%(K)=1000:GOSUB790:GOTO470
740 TG=TR:IFP=127ORP=255THEN930
750 POKESC+TG,32:GOSUB850:GOSUB790:GOTO470
760 POKEWF,0:POKEP1,P(N):POKEP3,V3:N=N+1
770 POKEVL,10:POKEAD,9:POKESR,0:IFN>3THEN N=0
780 POKEWF,21:POKESC+TR,91:POKECS+TR,1:GOTO470
790 FORJ=KTO 0 STEP-1:POKESC+TP%(J),32
800 POKEVL,5:POKEVL,0:NEXTJ:TH=0:RETURN
810 POKEWF,0:POKEP1,30:POKEAD,7:POKESR,0
820 POKEVL,15:POKEWF,129:FORJ=0TO15:POKESC+Q,63+J
830 POKECS+Q,7:NEXTJ:POKESC+Q,32:POKEWF,0:RETURN
840 REM** SWALLOWING BELOW
850 POKEWF,0:POKEAD,9:POKEVL,15:POKEWF,21
860 FORJ=KTO1 STEP-1:POKESC+TP%(J),81
870 POKEP1,10+8*J:POKEP3,10+J
880 POKESC+TP%(J),91:NEXTJ:RETURN
890 POKESC+TG,214:POKEWF,0:POKEAD,10:POKEVL,15
900 POKEP1,80+TG/10:POKEWF,21:FORJ=15TO0STEP-1
910 POKECS+TG,J/2:POKEP3,10+2*J:FORZ=0TO30
920 NEXTZ,J:POKESC+TG,32:POKECS+TG,1:RETURN
930 GOSUB890:GOSUB850:GOSUB790
940 POKESC+TG,32:IFS>HS THEN HS=S
950 POKEQA,0:PRINT"###SCORE"S"### HIGH"HS
960 S=0:K=0:PRINT"NEXT GAME?":GOTO430

```

LINE ANALYSIS FOR THE THING

100-140 Clear variables; locate character and color memory; turn sprites off; set sound-generation constants; turn all sound constants off; screen goes black; set sustain/release for sound.

150-170 PRINT title in white; DIMension arrays; delay; assign motion-increment arrays.

180-210 Assign blurb strings; PRINT blurb; set attack/decay.

220-240 An icy wind whistles through the trees using trigonometry.

250-280 A hole appears! Something is sucking the blurb into the hole like a piece of spaghetti.

290-340 The screen clears; PRINT the top screen line in green spaces; PRINT two lines of instructions in light blue; "You" in white, "the seed" in purple, "herbs" in green, and "the Thing" entirely in white.

350-390 Further keystroke instructions in blue; turn volume to medium.

400-420 Theramin-effect plays the "Thing Theme."

430-440 Prompt and GET a keystroke to continue.

450 Clears the screen and puts back the green border at top of screen.

460-470 Put the farmer at screen center; GET his X,Y position coordinates; await next valid keystroke.

480 GETs keystroke to move farmer; jumps to random ripening and Thing activation if no key was pressed.

490-510 Figure farmer's new position NP; increment move counter; switch to blue screen printing; toggle farmer to his alternate screen character; POKE out his old position, usually with a seed; PRINT out move count M; disallow moves off screen.

520 Jumps to endgame if moves are all used up.

530-550 Test for ripe herbs harvested; POKE farmer into new position; update score if herbs harvested; go back for next move.

560-580 Pick a random place on the screen twice; ripen any seeds found; jump back if less than three moves were made.

590-620 Jump ahead if the Thing is active; randomly activate the Thing; randomly select a note from the "Thing Theme"; replace that note with a random note.

630-650 Select a random place on the screen for the trapdoor to open; disallow occupied places; open the trapdoor.

660-750 Routine to advance the Thing from his trapdoor.

660-690 Determine X,Y coordinates of the trapdoor; assign jump vectors J%(); randomly select a vertical or horizontal move for the Thing.

700-720 TR is next screen position for the lengthening Thing; increment Thing-length counter K; set pitch of voice #3; retract the Thing if he is too long; look ahead for spaces and other Thing parts for next extension.

730 Retracts if the next position has a seed in it.

740-750 Target determined; if it's the farmer, jump ahead. Swallow the target (herbs); retract the Thing; go back for next player keystroke.

760-780 Note rotation for the "Thing Theme"; POKE in next section of the Thing; go back for next keystroke.

790-800 Retract Thing by POKEing him out backwards toward the trapdoor; make some noise; trapdoor also disappears. All in a subroutine.

810-830 Subroutine; explosion sound; rotate colors at the target location Q. On RETURNing, ripe herbs will appear.

840-880 Subroutine; Thing swallows something. It goes down his neck; neck restored at each vertebra; sounds made.

890-920 Subroutine; zork the farmer; pitches in voice #1 increment; farmer goes through eight colors, disappears.

930 Gobbles the farmer by calling appropriate subroutines.

940-960 Endgame; erase the target; increment all-time high score when necessary; PRINT scores; prompt for next game; go back to await a keystroke.

VARIABLE LIST FOR THE THING

In order of appearance:

SC and CS are first locations in screen character and color memory. QA and QB are keystroke and screen line counters in scratchpad memory.

P1, P3, AD, SR, VL, and WF are the memory locations used for sound generation.

P(1-4) are the pitches POKEd into sound for the "Thing Theme."

TP%(1-20) are the screen positions of the Thing as it advances.

M%(1-4) is the displacement array for screen motion.

A\$, B\$, and C\$ are strings making up the blurb.

PT is each pitch of the "Thing Theme" in turn.

L is the length of the blurb.

R is each last character of the blurb in turn as it is swallowed.

H,I,J, and Z are used as index counters in FOR-NEXT loops.

OC is the farmer's old character graphic (these alternate) or else the seeds he leaves behind.

M is the move counter, limited to 500 moves.

TH is a flag set to 1 if the Thing is in motion.

YP is your present position as the farmer.

NP is the farmer's next position.

YM is the farmer's present graphic character (your man).

X1 and Y1 are the X,Y coordinates of the farmer on the screen.

M\$ is the player's keystroke to move the farmer.

A is ASCII code for M\$ from 133 to 136.

MV = A - 133 from 0 to 3 to index the move increment M%(MV).

F = 40, always. This saves time and space on the LIST.

Q is a random screen position to test for seeds to ripen, the last seed position.

B is the contents of screen location SC + Q.

R,R% are also used as random variables here and there.

K counts how many lengths of the Thing are extended.

TR is each position of the Thing as it grows out.

X2 and Y2 are the X,Y coordinates of screen position TR above.

J%(0-1) are X,Y displacements for the random advancement of the Thing.

P is the contents of TR.

TG is a target position for zorking and swallowing.

N counts from 0 to 3 over and over to advance the notes of the "Thing Theme."



POCKET PUZZLE

Surely you remember those flat, square pocket puzzles kids had back in grade school. There were 15 numbered plastic squares that slid past each other on a board of 16 places. This always left one place open. You could hold it behind your back and scramble the order of the squares, the object being to slide them all back into numerical order.

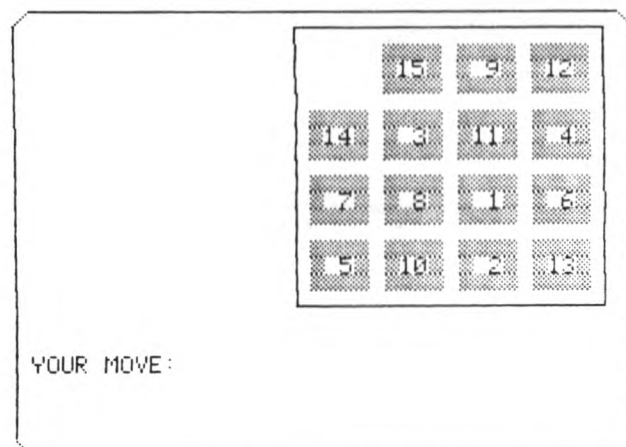
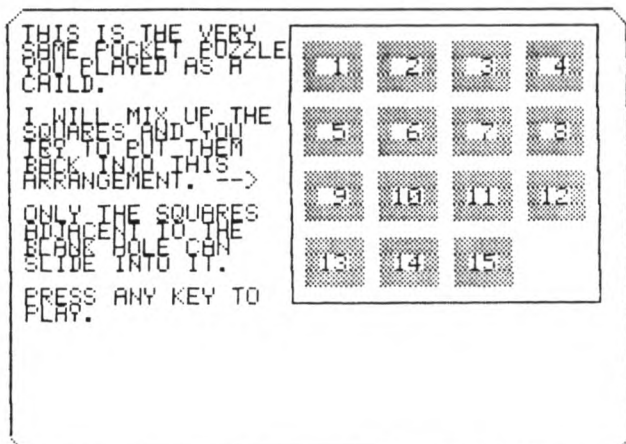
If you recall this childhood toy, the game program explains itself. The player is shown the puzzle board in perfect order along with instructions if he chooses. The program starts with a solved puzzle and asks if you want instruction. Assuming you don't, it immediately starts to scramble the board. The program will keep searching until it finds 80 legal moves and has made them.

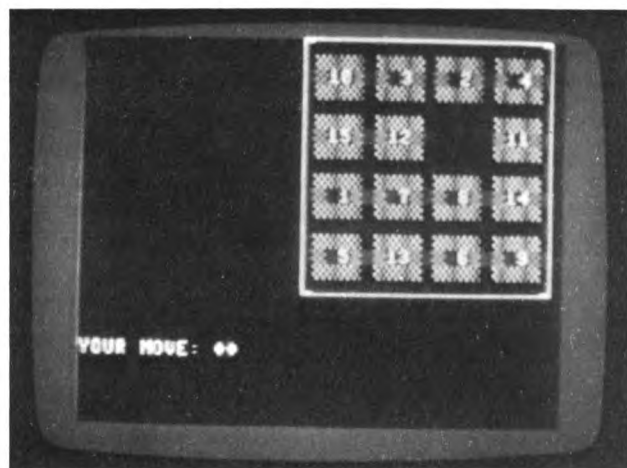
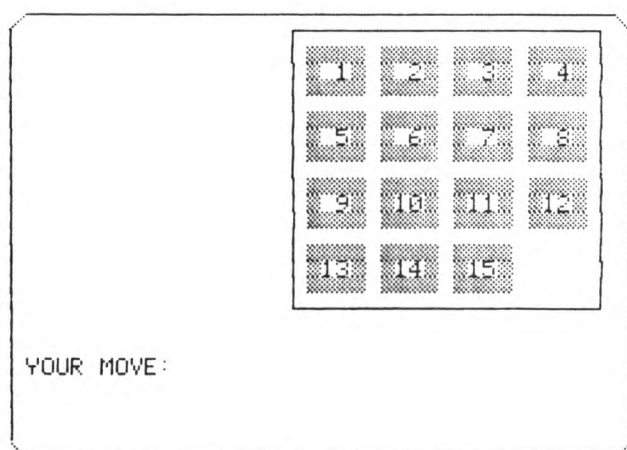
Then the user enters two-key moves representing the square he wants to push into the empty position. He does not need to indicate where to push his chosen square because there is only one place it can go; the machine will take care of that.

If the player tries to move a square when there is no adjacent empty position, the move is tossed out as illegal, and he enters another one. The game ends when or if the squares are in order with the blank square in the last (lower right) position. For one-digit entries like square #6, just press 6 and RETURN or some letter key. It is pretty simple to get the first two rows in order. The third is harder. The last row may fall into place or it may seem impossibly difficult.

In an earlier version of this game, the editors at Reston Publishing Company found a most embarrassing oversight. That version saved some program length by simply assigning the mixed up positions of the scrambled board at random. I had assumed, after experimenting, that any arrangement could eventually be twisted around to the correct solution. Not so! As it turns out, there are two mutually exclusive sets of arrangements, each with its own "solution"!

To put it another way, suppose you had the real plastic puzzle and all of the pieces fell out. If you put them back at random, there is a 50-50 chance that the 1 through 15 solution is totally impossible playing by the rules! The two solutions in the totally random case are the regular one, 1, 2, 3 . . . 13, 14, 15, and the unexpected one, 1, 2, 3 . . . 13, 15, 14!!





LIST #18: POCKET PUZZLE

```

1 REM*****
2 REM*POCKET PUZZLE      COMMODORE 64 *
3 REM*BY LARRY HATCH      11.17 *
4 REM*MENLO PARK, CA 94025    1983 *
5 REM*****
100 PRINT"POCKET PUZZLE *"
110 QB=214:POKE53280,9:POKE53269,0:POKE53281,0
120 DIMCN$(16,3),LO(16),LN(16),TB(16),RS(16)
130 FORI=1TO15:FORJ=1TO3STEP2:CN$(I,J)=" "

```

```

140 NEXTJ:CN$(I,2)=" " + RIGHT$(STR$(I),2) + " "
150 NEXTI:FORJ=1TO3:CN$(16,J)=" " :NEXTJ
160 E1$=" " :REM* 20 SP WIDE
170 E2$=" " :REM* INCL. EDGE
180 E3$=" " :FORI=1 TO 4
190 LN(I)=0:LN(I+4)=4:LN(I+8)=8:LN(I+12)=12
200 T=19+5*(I-1):TB(I)=T:TB(I+4)=T:TB(I+8)=T
210 TB(I+12)=T:NEXTI:CL$=" "
220 PRINT" ";TAB(18)E1$:FORI=0TO14:REM* TABLE
230 PRINTTAB(18)E2$:NEXTI:PRINTTAB(18)E3$:BL=16
240 FORI=1TO16:LO(I)=I:SQ=I:GOSUB630:NEXTI
250 PRINT"*****INSTRUCTIONS? ♦♦♦":GETY$:PRINTY$
260 IFY$="N"THEN400
270 IFY$<>"Y"THEN250
280 PRINT"THIS IS THE VERY"
290 PRINT"SAME POCKET PUZZLE"
300 PRINT"YOU PLAYED AS A":PRINT"CHILD."
310 PRINT"I WILL MIX UP THE"
320 PRINT"SQUARES AND YOU"
330 PRINT"TRY TO PUT THEM":PRINT"BACK INTO THIS"
340 PRINT"ARRANGEMENT. -->":PRINT
350 PRINT"ONLY THE SQUARES":PRINT"ADJACENT TO"
360 PRINT"THE BLANK HOLE":PRINT"CAN SLIDE."
370 PRINT:PRINT"PRESS ANY KEY TO":PRINT"PLAY."
380 GETZ$:IFZ$=""THEN380
390 PRINT" ";FORI=1TO20:PRINTCL$:NEXTI
400 A=1:B=162:C=15:FORI=1TO80:REM** SCRAMBLING
410 S=A+(PEEK(B)ANDC):GOSUB570
420 IFS=LM OR LG<1 THEN410
430 LT=LO(S):LO(S)=LO(BL):LO(BL)=LT:LM=S
440 SQ=S:GOSUB630:SQ=BL:GOSUB630:NEXTI:PRINT" ";
450 FORI=1TO10:PRINT" " :NEXTI
460 POKEQB,19:PRINT:PRINT"YOUR MOVE: ♦♦♦";
470 GETA$:PRINTA$:IFA$=""THEN460
480 GETB$:PRINTB$:IFB$=""THEN480
490 POKEQB,19:PRINT
500 PRINT" " 16 " " :PRINT" " 14 " "
510 M$=A$+B$:M=VAL(M$)
520 IFM<10RM>16THENPRINT" ?":GOTO460
530 S=M:GOSUB570:IFLG<1THENPRINT"ILLEGAL":GOTO460
540 LT=LO(S):LO(S)=LO(BL):LO(BL)=LT
550 PRINT" "
560 SQ=S:GOSUB630:SQ=BL:GOSUB630:GOTO460
570 X=LO(BL):REM*LEGAL MOVE FILTER
580 L(1)=X-4:L(2)=X+4:L(3)=X-1:L(4)=X+1
590 IF X/4=INT(X/4)THENL(4)=0
600 IF (X-1)/4=INT((X-1)/4)THENL(3)=0
610 LG=-10:FORK=1TO4:IFL(K)=LO(S)THENLG=I
620 NEXTK:RETURN
630 REM*PUT A SQUARE IN PLACE [SQ=SQUARE #]
640 L=LN(LO(S)):T=TB(LO(S)):POKEQB,L:PRINT
650 FORJ=1TO3:PRINTTAB(T)CN$(SQ,J):NEXTJ:RETURN

```

LINE ANALYSIS FOR POCKET PUZZLE

100-120 PRINT title in three lines at the left; assign QB; screen field black; turn sprites off; screen border brown; DIMension arrays.

130-150 Create 15 three-line blocks; top and bottom lines yellow, middle line blue with white numbers in the center. Create a 16th blank block (spaces).

160-180 Define puzzle borders: top, sides, and bottom.

190-210 Assign line number arrays for rows, tab arrays for columns, and CL\$ to blank out instructions and prompts.

220-230 PRINT in the puzzle borders.

240 Assigns square location array LO() into perfect order; PRINTs each square in including #BL = 16 for the blank square.

250-270 Prompt (insistently) for instructions; branch accordingly.

280-360 PRINT instructions.

370-390 Prompt to continue into game; await keystroke; blank out old instructions.

400-410 Set constants to speed up random selection of squares. Line 410 is a fast way to get random integers between 1 and 16.

420 Skips illegal and reversed moves for better scrambling.

430-450 Swap random square with the blank square; send cursor home; blank out old prompts.

460-480 The player's turn; prompt for his move; GET two keystrokes.

490-500 Send cursor down to clear the two-line prompts, etc.

510-520 Convert keystrokes to a number (1-16); disallow out-of-range entries.

530 Tests for legality of move; branches back if illegal. A branch depends upon some decision. A jump is unconditional for the purposes of these line descriptions.

540-560 Swap the two squares (chosen square and blank one) in the arrays; blank out prompts; call the subroutine to PRINT the squares in their new positions on the board.

570-620 Subroutine; test if a proposed move is legal. A move is legal if the square to be moved is directly adjacent to the empty position on the board.

580 Set up a test array of the four adjacent squares around the chosen square.

590-600 Disallow squares that are adjacent through two edges, i.e., not really adjacent at all.

610-620 Test if any of these adjacent positions contains the blank square; set legality flag LG accordingly.

630-650 Subroutine; put a square into place by printing in all three strings descending in line.

VARIABLE LIST FOR POCKET PUZZLE

In order of appearance:

CN\$(16,3) is the display of 16 blocks, each requiring three substrings to fill it out as a square.

LO(I) = 1 to 16, the location of the Ith block in the playing board.

LN(I) where I = 1 to 16 is the screen line number for the top of the Ith block. QB is poked to LN(I).

TB(I) is how far over to TAB when printing in the Ith block from CN\$(I, J) for J = 1 to 3.

RS(R) is an array of 16 flags indicating whether the Rth position is occupied on the gameboard or not.

I and J are loop index counters.

E1\$, E2\$, and E3\$ are the borders of the playing board.

T is a variable incremented by five to set up the horizontal tab array TB(I).

CL\$ is a string of 18 spaces to blank out the instructions.

SQ=1-16 is which square to fill in at subroutine on line 620.

Y\$ is the user response to the instructions prompt.

Z\$ is awaited after instructions to start actual play.

R = 1-16 randomly to mix up the squares at the start of play.

QB is the scratchpad memory location determining the next line to print on the C-64 screen.

A\$ and B\$ are the two keystrokes of each player move.

M\$ + A\$ + B\$ is the entire move from "x1" or "1x" to "16" where x is any dummy keystroke to get to two characters.

M is the value of M\$ from 1 to 16.

S = M pending approval of the move. Square S must border the empty square or it cannot move.

BL = 1-16 is the number of the blank unoccupied square on the board.

LT holds LO(S) while it is trading places with LO(BL).

SQ = S if the move is legal for the square-moving subroutine.

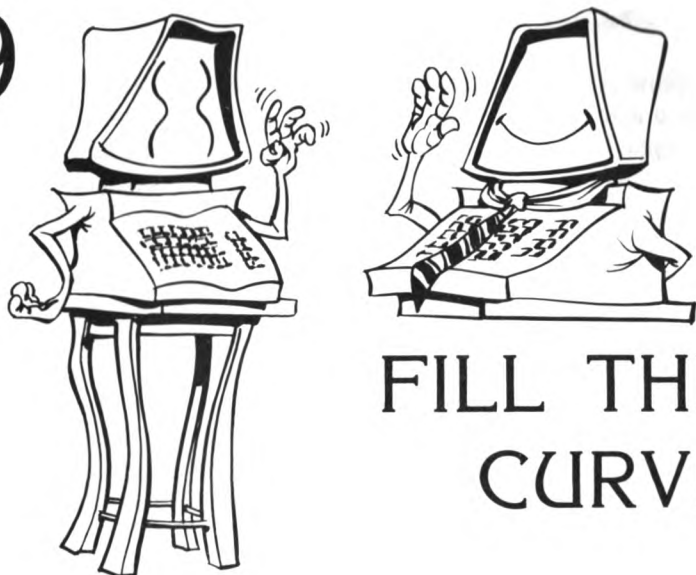
X = 1-16 is the location of the blank square.

L(I) where I = 1 to 4 itself equals 1-16 for the four squares adjacent to the blank square.

LG is a flag set positive to indicate a legal move.

L is the line number POKEd into QB for vertical (downward) screen printing position of a square.

T = TB(square #) to TAB the square over horizontally to the right.



FILL THE CURVE

This program fills in a mathematical curve that will pass through the points that you put onto the screen. Theoretically, there are an infinite number of curves that can pass through any number of points on a plane, but we are restricting ourselves to polynomials. To do this, we use the same Lagrange algorithm that we employed in the Polyform program earlier in the book. Instead of entering numbers, we run a sprite cursor around the screen, leaving little dots at will. The program will then run a nice-looking curve right through the points in high resolution graphics.

There are two features of this program that don't appear in the others. First, we "bit map" the entire screen to get dot-for-dot resolution for the curve. Second, to clear the 8000+ byte bit map, we use a very short and simple machine code subroutine, abandoning BASIC entirely for a few milliseconds. The same simple routine in BASIC would take about a minute. Both of these features deserve some explanation.

Your screen consists of 1000 bytes of character memory and an equal amount of color memory. We have been POKEing characters (and their colors) into the corresponding memory locations all along. These two memory areas are both arranged 40 columns wide by 25 rows deep. Each screen position requires one byte of information for character and another for color. Each character is defined by an 8 by 8 square of pixels, and this character occupies eight bytes (64 bits) of memory in your character-generator memory in ROM someplace.

Now we want control of every bit on the screen. That takes eight times as much memory. When we POKE a 1 into the fifth bit of byte 53265, we put the machine into bit-map mode. The highest four bits of location 53272 control where the bit-map memory will be. It will take eight bytes to control what used to be a character space, but we can control every dot.

Let's say our bit map starts at location 8192. The first 320 bytes control the top eight lines of pixels (dots) on the top of the screen. They are stacked on their heads like upright dominoes. If you POKE 8192,255, you turn on all eight bits. A line of dots, upended, should appear at the extreme left part of the screen top. POKE 8193,255 and the next line lights up to the right. The Commodore 64 *Programmers Reference Manual*, from which most of this information comes, is indispensable for these special features.

The bit map has to be cleared of "garbage" before we can use it. This is where the machine code comes in. I could call it assembly code, but I don't have a useable assembler in spite of a \$60 software purchase. The first three lines of DATA statements, the "loader" at line 240, and the SYS820 call at line 270 could have all been replaced with the BASIC line:

```
FOR I=8192 TO 16192:POKEI,0:NEXTI
```

You can fall asleep waiting for that to execute, so we resort to machine code. An assembly of the machine language subroutine would look like this:

Decimal	Hex code	Addr.	Assembly	Remarks
169,0	A9,00	0334	LDA #00	Load accumulator with zero.
162,32	A2,20	0336	LDX #20	Load X register with 32 decimal.
160,0	A0,00	0338	LDY #00	Load Y register with zero.
140,62,3	8C,3E,03	033A	STY \$033E	Store Y register at \$033E below.
141,0,32	8D,00,20	033D	STA \$2000	Store accumulator at address shown.
200	C8	0340	INY	Increment Y register contents.
192,0	C0,00	0341	CPY #00	Compare Y register to zero.
208,245	D0,F5	0343	BNE \$033A	Branch back if Y register not zero.
232	E8	0345	INX	Increment the X register.
142,63,3	8E,3F,03	0346	STX \$033F	Store X register into location 033F.
224,64	E0,40	0349	CPX #40	Compare X register to 64 decimal.
208,237	D0,ED	034B	BNE \$033A	Branch back if X register (64 dec.)
96,96,96	60,60,60	034D	RTS:RTS: RTS	Return to BASIC (3 times).

In assembly, the line numbers from 0334 are actual memory locations in the cassette buffer where we store the machine code. We load the accumulator with zero. The rest of the code alters the address after the STA

instruction at location \$033D in rotation until all 8000+ bytes addressed there are stored with the zero from the accumulator. We step the Y register from 0 to 255 and adjust the address. By doing this for X register = \$20 to \$40 (from 32 to 64 decimal) and putting this in to the MOST significant byte of the address, we step all the way through in a sort of nested loop.

A detailed explanation of this kind of programming is beyond the scope of this book. If you like speed, mind-boggling blinding speed, then you should get a good book on assembly (machine) code and start experimenting. This little subroutine is poorly written in some respects. Since it points to its own memory locations, it cannot be relocated in memory without some changes. The three RTS (ReTurn from Subroutine) instructions at the end remind me of Moses striking a rock in the desert with his staff. One strike should do it, but I want to be on the safe side. What if I miscalculated one of the branch instructions? The machine could run off into Never-Neverland, where the RESTORE key is ignored.

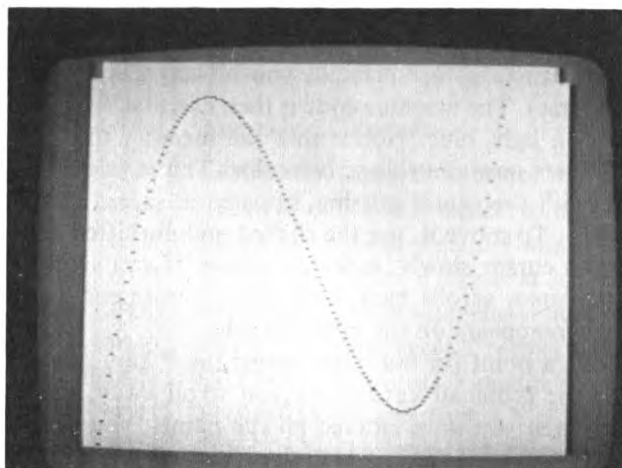
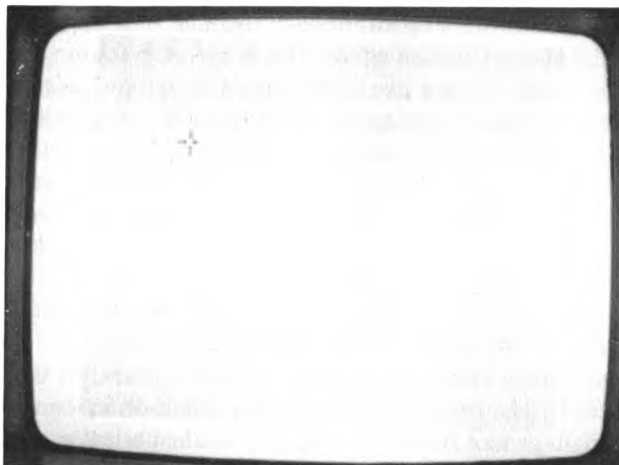
Arcade-type action games are all written in assembly code because of the speed required; there is just too much going on at once to use an interpreter language like BASIC. Logic and mathematical games are favored in these pages for this and other reasons. BASIC is good for quickly getting new program ideas running in an executable program. It is a great learning language, possibly the best.

After instructions are printed, you hit any key to continue (except STOP, of course). The machine code is then executed in a flash. The screen is POKEd in a light blue. Notice that the memory that usually controls screen characters now controls screen color. This is tolerably fast. A cross-hair sprite, with the center missing, appears at screen center. This is our pseudo-cursor. To move it, use the shifted and unshifted cursor keys. You will move this cursor slowly, a dot at a time. If you scroll off the top or bottom, the cursor scrolls back from the opposite end. If you scroll off either side, it reappears on the opposite side.

To mark a point for our curve, press the P key. A point will appear centered by the crosshair cursor. As you scroll away, the point remains stationary. When you have entered all the points, you press the F key to fill the curve. An 8-by-8-dot square at the upper left of the screen turns black if your curve begins beneath the visible screen area and white if we start out above it. Most curves will have some portions offscreen, especially at the sides. When the program has covered the width of the screen, both upper corners turn blue. As the points of the high-resolution display are calculated, little clicks are heard. To try another curve, it is not necessary to run the whole program. Instead, hit the R key and allow a few seconds to recolor the screen light blue. The cursor, which had vanished, will reappear for a new set of points.

The best-looking curves have a smallish number of points spread far

to the left and right but not too high or too low. A triangle of points usually makes a nice-looking parabola. It might make a good game to try to predict the shape of a curve from scattered points, before it is filled in.



LIST #19: FILL THE CURVE

```

1 REM*****
2 REM#FILL THE CURVE      COMMODORE 64*
3 REM#BY LARRY HATCH      (C) 1983*
4 REM#MENLO PARK CA      VRSN 11.18F *
5 REM*****
100 CLR:PRINT"☐"      FILL THE CURVE☐"

```

```

110 P1=54273:AD=54277:SR=54278:VL=54296:WF=54276
120 POKE53281,0:POKESR,0:QA=198:DIMX(99),Y(99)
130 PRINT"USE THE CURSOR AND SHIFT KEYS TO MOVE"
140 PRINT"THE POINTER SPRITE. TO SET A POINT,"
150 PRINT"PRESS THE P KEY."
160 PRINT"WHEN YOU HAVE SET ALL DESIRED POINTS,"
170 PRINT"THEN PRESS THE F-KEY TO FILL IN THE"
180 PRINT"MOST SIMPLE POLYNOMIAL CURVE WHICH"
190 PRINT"PASSES THROUGH ALL OF THESE POINTS."
200 PRINT"PRESS STOP AND RESTORE KEYS TOGETHER"
210 PRINT"TO RESTORE YOUR NORMAL SCREEN."
220 PRINT"PRESS ANY KEY TO CONTINUE..."
230 N2=2:N3=320:N7=7:N8=8:BM=8192:N5=53265
240 RESTORE:FORI=820TO848:READZ:POKEI,Z:NEXTI
250 FORI=896TO959:READZ:POKEI,Z:NEXTI:POKE2040,14
260 POKEQA,0:WAITQA,1:PRINT"J"
270 SYS820:POKEN5,PEEK(N5)OR32:POKE53269,1
280 POKE53287,0:POKE53248,0:POKE53249,0:N=0
290 POKE53272,PEEK(53272)OR8:REM BITMAP AT 8192
300 FORI=1024TO2023:POKEI,3:NEXTI:REM SET COLOR!
310 X=159:Y=99:GOSUB540:N=0:POKEVL,12:POKEQA,0
320 GETZ$:IFZ$=""THEN320
330 IFZ$="J"THENY=Y+1:GOSUB540:IFY>199THEN Y=0
340 IFZ$="I"THENY=Y-1:GOSUB540:IFY<0THEN Y=199
350 IFZ$="M"THENX=X+1:GOSUB540:IFX>319THEN X=0
360 IFZ$="N"THENX=X-1:GOSUB540:IFX<0THEN X=319
370 IFZ$="P"THEN GOSUB510:N=N+1:X(N)=X:Y(N)=Y
380 IFZ$<>"F"THEN320
390 POKE53269,0:FORX=0TO319STEP2:POKEWF,0
400 S=0:FORI=1TON:P=1:FORJ=1TON:IFJ=ITHEN420
410 A=X-X(J):B=X(I)-X(J):P=P*A/B
420 NEXTJ:S=S+Y(I)*P:NEXTI:IFS>199ORS<0THEN440
430 Y=S:GOSUB510:POKE1024,3:GOTO460
440 IFS<0THENPOKE1024,1
450 IFS>0THENPOKE1024,0
460 PT=99+65*SIN(S/2):POKEP1,PT
470 POKEAD,17:POKEWF,17:NEXTX
480 POKE1024,4:POKE1063,4:POKE53276,0
490 GETZ$:IFZ$<>"R"THEN490
500 POKE831,32:GOTO270:REM** FOR A NEW CURVE ***
510 CH=INT(X/N8):RW=INT(Y/N8):LN=YANDN7
520 BY=BM+RW*N3+CH*N8+LN:BI=N7-(N7ANDX)
530 POKEBY,PEEK(BY)OR(N2+BI):RETURN
540 SX=X+12:SY=Y+40
550 POKE53248,SXAND255:IFSX>255THENPOKE53264,1
560 IFSX<256THENPOKE53264,0
570 POKE53249,SY:RETURN:REM ASSEMBLY RTNE BELOW
580 DATA 169,0,162,32,160,0,140,62,3
590 DATA 141,0,32,200,192,0,208,245,232
600 DATA 142,63,3,224,64,208,237,96,96,96,0
610 DATA 0,0,0,0,0,0,0,0,0,0,8,0,0,8,0,0,8,0
620 DATA 0,8,0,0,28,0,0,8,0,0,65,0,15,227,240
630 DATA 0,65,0,0,8,0,0,28,0,0,8,0,0,8,0
640 DATA 0,8,0,0,8,0,0,0,0,0,0,0,0,0,0,0,0,0:REM SPRT

```

LINE ANALYSIS FOR FILL THE CURVE

100-120 Clear variables; PRINT title in white; assign sound constants; turn screen field black; sustain/release off; assign keyboard counter.

130-220 PRINT out instructions.

230 Sets numerical constants for algorithm speed later on.

240 Always RESTORE the DATA pointer to the beginning of DATA if that's what you want to READ. READ in machine-language code for a routine to clear the bit-map memory.

250 READs in the sprite-cursor image. All this data goes into the cassette buffer.

260 Waits for a keystroke; clears the screen.

270 Calls the machine code subroutine which "POKEs" all 8192 bytes of the bit map to zero; puts the machine in bit-map mode; turns on sprite #1.

280 Colors the sprite black; puts the sprite offscreen, x and y = zero; zeroes the point counter.

290-300 Start bit map at location 8192; POKE the whole screen cyan (sky blue).

310-320 Put the sprite cursor at screen center; set volume; await the next keystroke.

330-380 Test and branch for keystroke orders to move cursor left, right, up, or down; to "print" a point or to fill the curve.

390-480 Fill the curve routine.

390 Turns sprite off; loops through Lagrange algorithm 320 times for each pixel from left to right.

400-420 The Lagrange algorithm; branches for offscreen pixels.

430-450 Turn the offscreen indicators on and off as required; white = high; black = low; purple = all done.

460-470 Create and POKE in a pitch for sound generation.

480-500 Turn both upper corners of the screen purple to show we are done; keep sprite in single-color mode; await an R keystroke; adjust a counter in the machine code; go back for the next curve.

510-530 Fill in a point on the curve (see list of variables).

540-570 Adjust the sprite "most significant bit" position registers so we can use the whole screen.

580-600 Machine code to clear the bit map; stored in DATA statements so BASIC can load it simply.

610-640 Cursor sprite image bytes stored as DATA also.

VARIABLE LIST FOR FILL THE CURVE

In order of appearance:

QA is the scratchpad memory location of the keystroke counter.

N2, N3, N5, N7, and N8 are number constants. It speeds up repetitive algorithms to use lettered variables instead of numbers.

BM = 8192, the beginning of the bit map for our 8K-byte screen display.

I is used as an index counter in FOR-NEXT loops as usual.

N counts how many points are entered into the bit-mapped screen.

X, Y are first used to position the sprite cursor as we enter points.

Z\$ is each keystroke; to move the cursor, PRINT a point or fill the curve in.

X(n), Y(n) are arrays storing the coordinates of each point entered into the display.

X is used again to index points through the Lagrange algorithm and to put them on the screen.

Y is used again to determine the altitude of these points as we fill the curve.

S and P are sums and products used over and over in the Lagrange algorithm.

PT is a pitch, POKEd into P1 to show that the computer is calculating.

CH is the left-right "ordinary character position" that the next dot will appear in from left to right (0 to 39).

RW is the ordinary line number (0-24) that the dot will appear in.

LN (0-7) is which of eight horizontal rows of eight dots (pixels) the dot will appear in inside of an ordinary 8 by 8 character space.

BI (0-7) is which bit in line LN we wish to light up.

BY (8192-16192) is the byte in the 8K bit-map memory that we will have to POKE into to light up our pixel.

BM is 8192, which helps us find BY.

SX = X + 12 to make X = 1 appear at screen left.

SY = Y + 40 to make Y = 1 appear at the top of the screen.



BANDIT

Bandit is a slot machine simulation with improvements over certain earlier programs of this type. The three reels are simulated mathematically so that the symbols stay in position with respect to each other as they would on a real mechanical device. Three symbols are shown on each reel so the player can see what he almost won. There are sound effects, and the payouts are relatively generous, but as always, they favor the casino, your machine.

I saw a slot machine program published once that was so awful I hesitate to mention it. The display, if you can call it that, consisted of vertically printed words like "orange" or "lemon" so that the "reels" were on top of each other. On average, it must have paid off about one quarter of the wagers invested. The payoffs weren't even in pretend dollars or any ordinary currency but in some childish sounding play-money term. The intention was evidently to discourage the evils of gambling with pretend money!

This program has a few flaws as well. Sadly, due to the limit of eight sprites, we cannot use them for the reel displays. We would need 12 sprites, and BASIC is too slow to move them all at the same time with any realism. The colored character graphics make better stationary displays, but they are shown in pseudo-motion by means of green masks that are printed over with the stopped reels. Even so, we get an acceptably good effect.

I must also apologize for the square purple plums. If you have a black and white screen, the plums look like tombstones. There was no way to

create a good round plum, so I put the letter P in the middle to identify it. See the screen displays following.

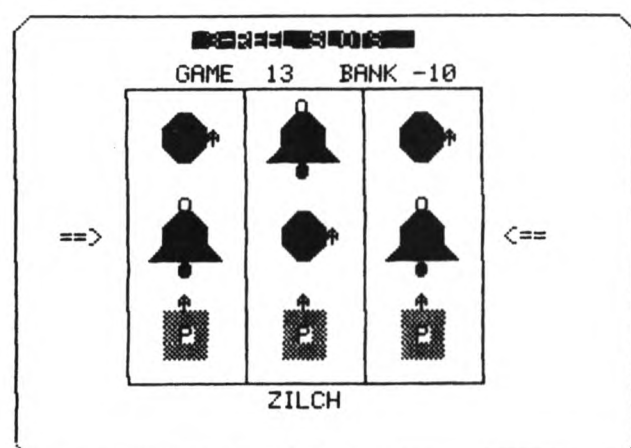
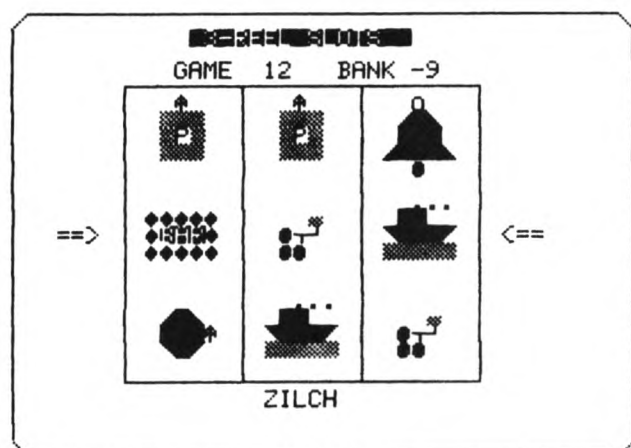
Play starts with the machine asking for a randomizing number. This number, along with how long it takes you to type it in, will randomize the random-number-generating algorithm in C-64 BASIC. Play is simple after that. The machine just keeps pulling its own lever, spinning the reels and keeping track of the money while the player watches. Since the player's coins are automatically deposited, he can let the machine gamble all night and see the results in the morning.

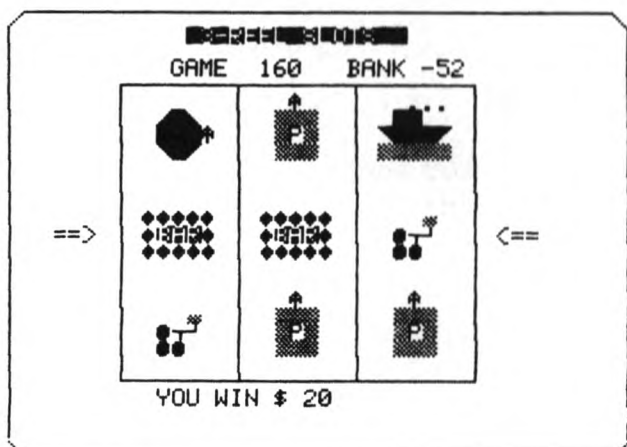
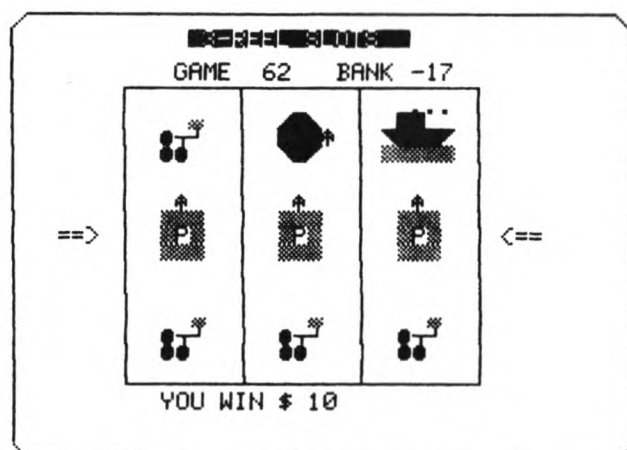
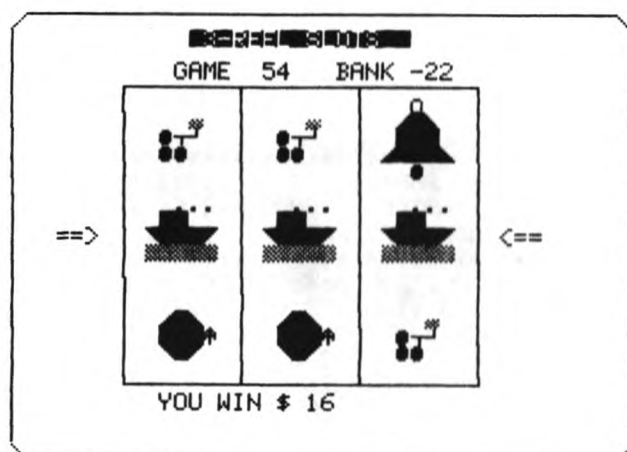
By using a reel of 30 positions, only one of which contains the BAR symbol, the jackpot of three bars seldom comes up. This enables a jackpot payout of \$250, which is higher than most casinos pay on a simple dollar machine. American machines generally pay higher prizes and have a lower house advantage in large legitimate casinos licensed by the state. British "fruit machines" are quite stingy by comparison. Illegal gambling operations are probably the worst of all. Slot machines are sometimes called one-armed bandits. That is the origin of this program's title.

According to gambling lore, the earliest slot machines were developed to relieve busy bartenders from the added task of shaking dice for drinks. These machines were produced in San Francisco and spread out from there. They come in many varieties, including a "Big Bertha" that is seven feet tall with reels the size of truck tires. One English fruit machine quacked at me like a duck. The way it gobbled 10 pence pieces, it should bark, meow, neigh, hiss, and dispense cigarettes.

In this simulation, the reels make a ratcheting sound as they stop, and if you lose, a small bang is heard. When you win some coins, the computer tries to quack like the English machine. It sounds more like the "caw-caw" of a crow or raven, however.

THIS IS A \$1.00 SLOT MACHINE.			
THE PAYOFFS ARE:			
BAR	BAR	BAR	\$250
BELL	BELL	BELL	300
BELL	BELL	BELL	300
SALT	SALT	SALT	150
SALT	SALT	SALT	150
PLUM	PLUM	PLUM	21
CHERRY	CHERRY	CHERRY	12
CHERRY	CHERRY	CHERRY	12
OTHER COMBINATIONS DONT PAY.			
PLEASE EXCUSE THE SQUARE PLUMS.			
A RANDOMIZING NUMBER PLEASE? 593			





LIST #20: BANDIT

```

1 REM*****
2 REM* ONE ARMED BANDIT (C) 1983 *
3 REM* BY LARRY HATCH COMMODORE 64 *
4 REM* MENLO PARK, CALIF 94025 11.21*
5 REM*****
100 CLR:PRINT"  3-REEL SLOTS  "
110 QA=198:QB=214:V1=54273:V3=54287:VL=54296
120 AD=54277:WF=54276:POKE53280,9:POKE53281,0
130 POKE53269,0:DIMA$(200,3),F$(6,5),A$(31,3),R(3)
140 PRINT:PRINT" THIS IS A $1.00 SLOT MACHINE."
150 PRINT:PRINT" THE PAYOFFS ARE:"
160 PRINT"♦BAR♦ ♦BAR♦ ♦BAR♦ .....$250"
170 PRINT"♦BAR♦ ♦BAR♦ -ANY- ..... 20"
180 PRINT"BELL BELL BELL ..... 30"
190 PRINT"BELL BELL -ANY- ..... 5"
200 PRINT"SHIP SHIP SHIP ..... 16"
210 PRINT"SHIP SHIP -ANY- ..... 3"
220 PRINT"PLUM PLUM PLUM ..... 10"
230 PRINT"LEMONS DONT PAY !! .....ZIP"
240 PRINT"CHERRY --- --- ..... 3"
250 PRINT"CHERRY CHERRY --- ..... 5"
260 PRINT"CHERRY CHERRY CHERRY ..... 12"
270 PRINT:PRINT"OTHER COMBINATIONS DONT PAY."
280 PRINT"PLEASE EXCUSE THE SQUARE PLUMS."
290 RESTORE:GOSUB770:T1=TI:POKEQA,0
300 INPUT"A RANDOMIZING NUMBER PLEASE";Z
310 T2=TI-T1:X=RND(-(Z+T2/30)):T1=T2
320 PRINT"  3-REEL SLOTS  "
330 PRINT"GAME "G;" BANK $"B
340 G=G+1:B=B-1:REM*NEXT PULL
350 B=(INT((B*20)+.0001))/20:I=0
360 PRINT"GAME "G;" BANK $$$$B"
370 FORI=1TO300:Z1=1:Z1=0:NEXTI:I=0
380 GOSUB 700:FORI=0TO300:Z1=1:Z1=0:NEXTI
390 IFG/97=INT(G/97)THENT2=-ABS(TI-T1)/9873:T1=T2
400 REM -REEL RANDOM SUBROUTINE-
410 FORI=1TO3:R=INT((RND(Z)*30)+1):RP=I
420 PRINT" :F=R:IF F=31THENF=1
430 S=R*6-4:T=8+8*(I-1):R(I)=A(F,I)
440 FORJ=STOS+16:PRINTTAB(T)A$(J,I)
450 POKEVL,(J-S)/2:POKEVL,0:NEXTJ:POKEVL,15
460 POKEVL,0:NEXTI:N=0:IFR(1)<>6THEN490
470 N=3:IFR(2)=6THENN=5:IFR(3)=6THENN=12
480 IFN>0THEN640
490 IFR(1)<>R(2)THEN N=0:GOTO600
500 IFR(1)=1THENN=20:IFR(3)=1THENN=250
510 IFN>0THEN640
520 IFR(1)=2THENN=5:IFR(3)=2THENN=30
530 IFR(1)=3THENN=3:IFR(3)=3THENN=16
540 IFN>0THEN640
550 IFR(1)<>R(3)THEN N=0:GOTO600
560 IFR(1)=4ANDR(3)=4THENN=10

```


- 290-300 Set DATA pointer to first item; GOSUB string assignment subroutine; note the time (in jiffies) as T1; INPUT a randomizing number.
- 310-330 Randomize the RND function; clear the screen with a new title and statistics display.
- 340-360 Start the main loop; increment game counter; decrement the "bank" rounding off; PRINT out game and bank statistics.
- 370-380 Delay loop; call subroutine to print in window with green pseudo-spinning reels; delay loop again.
- 390-400 Occasionally, re-randomize the RND function; REM is a non-executable remark.
- 410-420 Find three random positions for the three reels to stop at; make end of reel equivalent to its beginning.
- 430-460 Calculate a stop spot S on each reel; calculate T to TABulate the display into the proper reel; assign R(I) the number of the symbol in each reel center; PRINT in each stopped reel in turn, making ratchet sound.
- 460-480 Check first reel for cherries; branch ahead if none; count cherries; skip to payout if any cherries.
- 490-510 If first two symbols don't match, then jump ahead. If matched symbols are two or three jackpots, then assign prizes and jump to payout routine.
- 520-530 Test for 2 or 3, bells or ships, assigning prizes accordingly.
- 540-550 Test for any prizes so far and branch; test if outer reels match, if not, no prize, so branch.
- 560-570 Test for three plums (prize) or any lemons (no prize) and branch.
- 580-590 Test if anything won so far and branch; ELSE, blank out prompts and go back for next spin. *Note:* Commodore BASIC does not support the ELSE statement. Lines 580 and 590 should not execute; they are a safety net in case of errors.
- 600-630 Player loses the spin; delay loop; shoot off a "bang"; PRINT "zilch" over any previous payout messages; go back for next pull.
- 640-690 Announce a win; serially add to the bank and make a crow sound for each coin won; update bank display; go back for the next pull.
- 700-760 Subroutine; set up the spinning-reels machine window display; put in the pay arrows.
- 770-940 Subroutine; assign color strings to make up the machine

symbols; READ the reel symbol order for each reel from DATA. Assign the reel arrays from the symbol arrays using the DATA numbers to put them into order.

950-1000 Numerical DATA having the symbol numbers in order for each reel to simulate a mechanical device.

VARIABLE LIST FOR BANDIT

In order of appearance:

A\$(200,3) is 200 lines times three reels worth of slot machine reel symbols. Each symbol requires six lines including blank lines for spacing. F\$(6,5) is the six lines of each type of symbol times five different types.

A(I,J) keeps track of which symbol is in the Ith position on the Jth reel.

R(I) records which symbol came up randomly on each reel.

V1, V3, VL, AD, and WF are the standard pitch, waveform, attack/decay, and volume POKE memory locations.

T1 is set to the constantly incremented system clock TI to help randomize.

X, Z, T1, and T2 are used to randomize the RND(1) function random numbers each time the game is played.

G is the play number, it counts pulls of the lever.

B is the player's bank in dollars. He starts with a zero balance.

I, J, and K are FOR-NEXT loop index counters.

Z1 is a dummy variable to lengthen a delay loop.

R randomly = 1 to 30 to stop the reels on some position.

RP records which reel was spinning for diagnostics only.

F is the position of the last reel stopped from 1 to 30.

S = 2 to 176 for the first line of A\$(S, I) to print out, displaying the stopped reel.

T = 8, 16 or 24 to TAB the display over horizontally for each reel.

N is the number of dollars (bucks) paid out on winning combinations.

BD is the player's bank plus the increment K to advance the bank display in units.

Z is used as a dummy delay loop index.

W is the winnings. W=N.

E\$ is the blank line used to set up the reel display array A\$ (I, J).



The title *Entrigma* is taken from the German Enigma cypher machines used by the axis powers during World War II. While the codes generated by these machines were cracked by the Allies, it required tremendous effort and resources. This program should protect personal transmissions or private diaries, etc., from casual snooping.

A similar method using a keyword was used up to the nineteenth century. If your keyword was, say, "HOUSE," then you would write HOUSEHOUSEHOUSE . . . below your plain language message, having removed the spaces and all punctuation. Then you simply added the letters, and your code was the sum.

For example, let's say your first plaintext word is "order." The letter O is 15 for the 15th letter of the alphabet, while H from House is 8. Adding, the sum is 23; the first letter of your code is the 23rd letter W. Similarly, R from oRder (18) plus O from hOuse (15) equals 33. Whenever the sum of two letters exceeds 26, we subtract 26 from it. Thus, $15 + 18 = 33$. Subtracting 26, we have 7, and the second letter of the code is G.

The plaintext word "order" becomes WGYXW. Note how the letter W stands for two different letters, O and R, from the plaintext. In this method, any letter can stand for any other letter at any time. The receiver of the message simply subtracts the repeating key from the code, adding 26 when the difference is less than one. Unfortunately, a retired Prussian Army major named Friedrich Kasiski found a simple statistical method to uncover the plaintext and the key. The lesson learned is not to use a repeating key!

The only theoretically unbreakable method is to use an unending key of completely random letters and never, never to use the same section of that key twice.

This poses a problem—how to send an identical copy of the same random series to whomever receives the message! You can't send it by the same channel you send the code in, certainly. It has been demonstrated that such a thoroughly random key causes a code so obscure that any possible decoding of the message is possible. Completely contradictory decodings may occur, and the codebreaker has no way to determine which message is the intended transmission!

Here we use a pseudo-random number generator for our never-ending key. The trigonometric functions at the end of the program will produce a scrambled mess. Since they are published here, you will want to change at least the coefficients of these functions, if not the functions themselves. Prime numbers help avoid repeating series. Always use arguments (the part in brackets) that have no common factors. The sum of 25 sine waves with identical arguments is a single sine wave!

Print out the output of the key generator at length until you are satisfied it is pseudo-random in appearance and avoids obvious patterns. One good test message is the same letter repeated 200 or 300 times. There should be only random repetition in the cypher text.

The program will start by asking whether you are coding or decoding. You code when you wish to send a secret message. Next you will be prompted for a code setting number. You can start with one for your first message, but you can never use that part of the pseudo-random sequence again. There is no faster way to blow a cypher than to use the same key twice.

Write down the ending number someplace. This will be one less than your starting code setting number for the next transmission. In two-way communication, your starting number is one more than the ending number or the last message sent or received. Once in a while, it is good to skip ahead in the sequence without using small parts of it. This causes discontinuities that help foil cryptanalysis. Note that we use the ABS function to make all key numbers positive. This causes more discontinuities in the pseudo-random key.

After entering the code setting number, you are prompted to enter your plain-language message. A diamond cursor appears at upper screen left, and letter keys appear where the cursor shows. Keys other than the 26 letters of the alphabet are ignored, with the exception of the DELEte key and the STOP key. The STOP key stops the program and puts you into the immediate mode. The DELEte key erases the last letter entered and backspaces the diamond cursor to correct errors. Numbers have to be spelled

out "one owe nine," and codes must be prearranged for punctuation. A double XX is sometimes used to end a sentence, for instance.

At the end of your message, you hit the asterisk key. This tells the program to start encoding. Your original message is reprinted in neat five-letter columns in the best cloak and dagger tradition. If this is difficult to read, don't feel bad. The Nazis had the identical problem, and so do some other nasty types. This screen printing in columns is slow because the cyphertext is being generated at the same time. Next, the cyphertext is rapidly PRINTed to the screen for copying into a diary or letter.

If you have a printer, then remove the REMs and asterisks at the beginnings of lines 380 and 430. This should give you instant hard copy. Check your printer manual to verify device and channel numbers and to ensure that the Commodore printer syntax is still the same. You have to spell out the word PRINT. Don't abbreviate with a question mark token. The older Commodore printers, at least, will not accept this token. If your printer is as noisy as mine, you may not want to use it at all.

To verify that there are no garbles in your cyphered message, you have the option to verify to original. This decodes the message back to plaintext (sometimes called clear) for comparison to the original.

Anyone having an identical version of this program can decode a message just as easily. All he needs to do is press D for Decode when prompted and to type in the garbled cyphertext using the same diamond cursor. He will need to know the starting code setting number, of course. As soon as he hits the asterisk, the program works in reverse, subtracting out the same pseudo-random key that was put in originally. Actually, the only part of the program that needs to be identical is the key algorithm after line 620. You could convert this program to run on say, an Apple® or an IBM®PC and send messages between users with entirely different machines. It would help to study the line analysis after the LISTing to make such conversions.

```

ORDER RECIEVED X AM SENDING TWO CATS A B
TITLE OF SCOTCH AND A BAR OF LIBRARY PAS
TE X WHAT ARE YOU PLANNING TO DO X*
ORDER RECIEVED X SEND INGTW OCATS
ABOUT LEAFS COUCH ANDRA ANING TODOX
YPAST EXXHA TAREY OUPLA ANING TODOX

OUTPUT FOLLOWS:
DWNYO ZRIZZ CCBKZ WHQNX KEYEB RAUOK
IDBZT MZEEF BILLB AHASE CQBOT KINFT
DRLCT VOZAM VZATA AVASE GJFAD TZSYO

SETTINGS: FIRST 8402, LAST 8492
VERIFY TO ORIGINAL? Y
ORDER RECIEVED X SEND INGTW OCATS
ABOUT LEAFS COUCH ANDRA ANING TODOX
YPAST EXXHA TAREY OUPLA ANING TODOX

```

THE PLAN IS TO GET THE LANDLORD BOMBED A
 NO. 6. THE DEPT. IS TO HIS OVERCOAT AND CAL
 THE FIRE DEPARTMENT X WATCH THE EVENIN
 G NEWS.
 THEPL ANIST OGETT HELAN DLORD BOMBE
 DANDG LUETH ECATS TOBIS OVERC OBTIN
 EVEN INGNE NSX

OUTPUT FOLLOWS:

GIUEG LYMMB QIMXU JCLSI XWRKU LSGLR
 LZZND DUMSZ TIGBK YDWA BJETH SPESQ
 DSSGR NGUST BRUD JSBEX LMOLR JEPSQ
 AOZOR LYBGA PAD

SETTINGS: FIRST 8493, LAST 8596
 VERIFY TO ORIGINAL? Y

THEPL ANIST OGETT HELAN DLORD BOMBE
 DANDG LUETH ECATS TOBIS OVERC OBTIN
 EVEN INGNE NSX

LIST #21: ENTRIGMA

```

1 REM*****
2 REM* ENTRIGMA  ENCRYPTION SYSTEM *
3 REM* LARRY HATCH      COMMODORE 64 *
4 REM* MEMORY 6000-7000 POKES  11.23*
5 REM* MENLO PARK, CA 94025    1983 *
6 REM*****
100 CLR:PRINT"ENTRIGMA ENCRYPTION SYSTEM"
110 IN=6000:OT=6500:REM* RELOCATABLE MEMORY
120 QA=198:QB=214:POKE53281,0:REM*BLACK SCREEN
130 SC=1024:PRINT"CODE OR DECODE (C OR D) ";
140 POKEQA,0:WAITQA,1:GETZ$:PRINTZ$
150 SZ=1:IFZ$="D" THENSZ=-1:GOTO170
160 IFZ$<>"C"THENPRINT"":GOTO130
170 INPUT"CODE SETTING NUMBER";SN:SN=INT(SN+.1)
180 Q=SN:PRINT":POKEQB,20:PRINT
190 PRINT"ENTER DATA; END WITH ASTERISK *";
200 POKEQA,0:WAITQA,1:GETC$:AC=ASC(C$)
205 IFC$="*"THEN250
210 IF AC=20 THENPRINT"  ";N=N-1
220 IF AC=32 THENPRINT"  ";N=N+1
230 IFAC<65ORAC>90THEN200
240 PRINTC$";N=N+1:GOTO200
250 L=N+1:J=0:FORI=0TOL:CY=PEEK(SC+I)
260 IFCY>0ANDCY<27THENJ=J+1:POKE IN+J,CY
270 NEXTI:NI=J:POKEQB,20:PRINT
280 PRINT"          36
290 POKEQB,1+NI/40:PRINT:FORJ=1TONI
300 AZ=PEEK(IN+J)+64:PRINTCHR$(AZ);
310 IFJ/5=INT(J/5)THENPRINT" ";GP=GP+1
320 IFGP>5THEN PRINT:GP=0
330 GOSUB620:OZ=PEEK(IN+J)+K*SZ:Q=Q+1
340 IFOZ>26THENOZ=OZ-26:GOTO340

```

```

350 IF O% < 1 THEN O% = O% + 26 : GOTO 350
360 POKE OT + J, O% : NEXT J : PRINT
370 GP = 0 : PRINT : PRINT "OUTPUT FOLLOWS:" : PRINT
380 REM ***** OPEN1, 4 : CMD1 : REM* PRINTER OPTION
390 FOR J = 1 TO N1 : B% = PEEK(OT + J) + 64 : PRINT CHR$(B%) ;
400 IF J / 5 = INT(J / 5) THEN PRINT " " : GP = GP + 1
410 IF GP > 5 THEN PRINT : GP = 0
420 NEXT J : PRINT : PRINT "X SETTINGS: FIRST " SN " LAST " Q
430 REM ** PRINT#1 : CLOSE1 : REM* PRINTER OPTION
440 PRINT "VERIFY TO ORIGINAL? ♦||" : POKE QA, 0
450 WAIT QA, 1 : GET Z$ : PRINT Z$ : IF Z$ = "N" THEN 560
460 IF Z$ <> "Y" THEN PRINT "J" : GOTO 440
470 REM* VERIFY TO CLEAR
480 GP = 0 : Q = SN : PRINT : FOR J = 1 TO N1
490 GOSUB 620 : Q = Q + 1 : B% = PEEK(OT + J) - K * S%
500 IF B% < 1 THEN B% = B% + 26 : GOTO 500
510 IF B% > 26 THEN B% = B% - 26 : GOTO 510
520 PRINT CHR$(B% + 64) ;
530 IF J / 5 = INT(J / 5) THEN PRINT " " : GP = GP + 1
540 IF GP > 5 THEN PRINT : GP = 0
550 NEXT J
560 IF S% <> -1 THEN 610
570 PRINT : PRINT "TRY ANOTHER CODE SETTING? ♦||" ;
580 POKE QA, 0 : WAIT QA, 1 : GET Z$ : PRINT Z$
590 IF Z$ = "N" THEN 610
600 INPUT "NEW SETTING"; SN : Q = SN : PRINT "J" : GOTO 290
610 PRINT : END
620 K = SIN(23 * Q) : REM- KEY GENERATOR
630 K = K + SIN(37 * Q) + SIN(47 * Q) + SIN(53 * Q) + SIN(67 * Q)
640 K = K + SIN(79 * Q) + SIN(83 * Q) + SIN(89 * Q) + SIN(97 * Q)
650 K1 = INT(ABS(50 * K)) : K = K1 - 26 * INT(K1 / 26) : RETURN

```

LINE ANALYSIS FOR ENTRIGMA

100-120 Clear variables; assign first input/output memory locations; assign keyboard and screen-line control constants; screen goes black.

130-140 Assign screen character memory constant SC; GET key-stroke for coding or decoding.

150-160 Set sign variable S% (1 = code, -1 = decode); disallow irrelevant keystrokes.

170-180 Prompt/INPUT index number of the key generation algorithm.

190-230 GET the input text (whether in cypher or plain); test for end of text (*); backspace and decrement character counter if delete key pressed; advance cursor and increment character counter for spaces; disallow non-alphabetic characters otherwise.

240 PRINTs character; advances pseudo-cursor; increments character counter N; goes back for next keystroke.

250-280 PEEK each input-text character from the screen C%; filter out everything but letters; POKE C% into input memory from location 6000 up; end loop; clear old prompts.

290-320 Set real cursor (invisible) below the input text; PEEK input text from input memory and print it out on screen in neat five-letter columns.

330-350 GET the key algorithm's Qth value K; add key to Jth character in plaintext to get 0%; increment Q; keep 0% between 1 and 26.

360-380 POKE each 0% into output memory from location 6500 up; announce output; allow printer option.

390-420 Print output to screen in five-letter columns; announce final key-generator settings.

430-460 Close printer file (optional); prompt/GET keystroke for verify option.

470-550 Verify option; same as code/decode routine, but line 490 subtracts the already signed keys K*S%.

560-590 Jump to end if not decoding; allow option to try decoding with a different starting key index number; branch to end if option refused.

600-610 INPUT new key-index setting; jump back to use it.

620-650 Key generation algorithm subroutine; K becomes the sum of oddball sine functions of Q, all with different periods of oscillation; K1 is the absolute (positive) value of K; K becomes K1 Modulo 26, i.e., from 0 to 25 by subtracting out all whole 26s.

VARIABLE LIST FOR ENTRIGMA

In order of appearance:

IN (=6000) is the first memory location used to store the processed input text, plain or code.

OT (=6500) is the first memory location to store the coded or decoded output text.

Z\$ is any one-key user "input" to select options.

S% = 1 or -1 to add or subtract the key number from the alphabetic number of the text.

SN is the starting key index number that the user puts in to avoid repetition of sections of the running key.

Q=SN at first; used to count up to the ending code number.

C\$ is each character of input text entered in turn.

N is a character counter as text is typed in.

$L = N + 1$ is the count of all valid letters plus the asterisk at the end.

J increments up to N to index the characters in memory.

I is a FOR-NEXT loop index counter.

C% is the screen memory PEEK value of each text character in turn.

NI is the final length of the text input in alphabetic letters.

A% is the ASCII code number for each letter stored in memory after IN.

B% is the character number of output plus or minus the key for output.

GP is a group-of-five counter used to drop a line after six groups.

O% is each character number after its particular key is added or subtracted from the original letter.

Q = Q% the key index, as used in the key-generation subroutine.

K and K1 are intermediate numbers used in key generation.

K is the final key, changing for each letter in the message, to be added to or subtracted from the ASCII code for that letter.



Decahex is a decimal to hexadecimal number converter. Try saying that backwards! Decahex also converts hexadecimal numbers to decimal. A program like this is not really a game; it is better called a "utility." Utilities are programs that help to write and debug game and other programs under development.

Hexadecimal ("hex") numbers are much closer in nature to the binary numbers used internally by your computer. They work like decimal numbers, but instead of ten digits (zero to nine), there are 16 (zero to F), with F equal to 15 in decimal. After that, you have to carry a digit and write 10, 11, . . . 19, 1A, 1B, 1C, 1D, and 1F. Then you continue with 20, 21 . . . 2F.

This goes on through 90, A0 . . . F0 . . FF, after which we carry again to \$100. We sometimes write \$100 or 100 with a little 16 written underneath to differentiate \$100 from regular decimal 100. \$100 equals 256 decimal. The first screen character memory location is 1024. In hex, this number is \$400. The entire memory map of the Commodore 64 is between \$0000 and \$FFFF = 65535. 65535 is 16 to the 4th power minus one. We always count from zero, remember.

Memory maps of computers are often written in hex, which is inconvenient when you can only PEEK and POKE in decimal. Machine language monitors and assembler programs usually converse entirely in hex. For this reason, I put together Decahex to translate numbers in either direction.

When you run the program, it asks which way you are converting. If you press the Y key, the machine will convert decimal numbers to their hex equivalents. Use any other key and the machine assumes you are converting hex to decimal.

Then you type in your number as INPUT, and the machine translates. As long as you put in numbers of the correct type, it keeps on translating. If you are converting from hex to decimal and you press a Y key, the machine will switch to the opposite conversion. At any time, you can press X or Y and change the direction of conversion.

Negative numbers are not allowed; different microprocessors handle them differently. Any character can be part of a hex number here, but the machine will translate any digit not between 0 and F as a zero. When a strange character, like a W or a space, is found in a hex input, an error message will show up in yellow. Similarly, the program will not permit decimal number inputs less than one as these gum up the algorithm. A string of question marks will replace the usual eight digits of hex output on the screen in this case.

The range of numbers is from 1 to over 4 billion, more than enough to decode memory addresses. This is 0000 0000 to FFFF FFFF in hexadecimal.

```
ENTER HEXADECIMAL # ? FD16
```

```
DECIMAL= 64790
```

ENTER DECIMAL NUMBER? 4294967295

HEX # = FFFF FFFF

ENTER DECIMAL NUMBER? -427.65

HEX # = ???? ????

LIST #22: DECAHEX

```

1 REM*****
2 REM*DECAHEX          COMMODORE 64 *
3 REM*BY LARRY HATCH   (C)    1983 *
4 REM*MENLO PARK, CALIF    11.23*
5 REM*****
100 CLR:QB=214:POKE53281,0:VL=54296
110 PRINT"DECAHEX TO DECIMAL CONVERTER"
120 PRINT"FOR HEX TO DECIMAL CONVERSION PRESS X"
130 DIM HD(7),H$(15):FORI=0TO9:H$(I)=CHR$(I+48)
140 NEXTI:FORI=10TO15:H$(I)=CHR$(I+55):NEXTI

```

```

150 PRINT"FOR DECIMAL TO HEX CONVERSION PRESS Y"
160 PRINT"YOU CAN SWITCH AT ANY TIME."
170 GETC$:IFC$=""THEN170
180 PRINT"J":IFC$="Y"THEN270
190 INPUT"ENTER HEXADECIMAL #";H$:L=LEN(H$):DC=0
200 E$="":FORI=0TOL-1:D$=MID$(H$,L-I,1)
210 D=VAL(D$):IFD$="0" OR D<>0 THEN240
220 D=ASC(D$)-55:IFD$="Y"THENI=L-1:NEXTI:GOTO270
230 IFD<10ORD>15THEND=0:E$="ERROR ↑"
240 POKEVL,15:DC=DC+D*16↑I:POKEVL,0:NEXTI
250 POKEVB,5:PRINT:PRINT"J"
260 PRINT"DECIMAL="DC" ";E$":GOTO190
270 FORI=0TOL:HD(I)=0:NEXTI:Z8=0:Z9=0
280 INPUT"ENTER DECIMAL ... #";T$:L=LEN(T$)
290 FORI=1TOL:M$=MID$(T$,I,1):IFM$="X"THENZ9=9
300 A=ASC(M$):IFA<48 OR A>57THENZ8=8
310 NEXTI:IFZ9=9THENPRINT"J":GOTO190
320 T=INT(VAL(T$)):P=7:IFZ8=8 THEN390
330 S=16↑P:IFS>T THENP=P-1:GOTO330
340 T=T-S:HD(P)=HD(P)+1:IFP>0ANDT>0THEN330
350 IFT>0THEN340
360 PRINT"HEX # = ";:FORI=7T00 STEP-1
370 POKEVL,15:PRINT H$(HD(I)):IFI=4THENPRINT" ";
380 POKEVL,0:NEXTI:PRINT":GOTO270
390 PRINT"HEX # =  ???? ????":GOTO270

```

LINE ANALYSIS FOR DECAHEX

100-110 Clear variables; assign print-line and sound constants; screen goes black; clear screen and PRINT title in white.

120-140 First instructions; assign H\$(i) 16 hexadecimal characters.

150-180 More instructions; await keystroke; branch; disallow irrelevant entries.

190-250 Hexadecimal to decimal conversion routine.

190-210 INPUT hexadecimal string H\$; examine each character in H\$; check for digits 0 to 9; branch ahead if one found.

220-230 Evaluate digits A through F; signal an error if other characters are found.

240 Makes a noise; increments decimal value by the hex digit examined to the appropriate power of 16.

250-260 PRINT out final decimal value; jump back for the next number.

270-290 Decimal to hexadecimal conversion routine.

270-280 Zero conversion variables; prompt for decimal input T\$; GET length of T\$.

290-310 Examine each character of T\$ (as M\$) to be sure it's a digit from 0 to 9; set flags Z8 and Z9 accordingly and branch.

320 T is the numerical value of T\$; sets conversion variables.

330-350 Subtract successively smaller powers of 16 from T; increment the hex digits HD(p) accordingly; branch back until T is used up.

360-380 PRINT out the hexadecimal result; break into two groups of four hex digits each; blank prompts and go back for next number.

390 Error display shown if the decimal number string contains garbage or spaces.

VARIABLE LIST FOR DECAHEX

In order of appearance:

QB is the memory location determining the next line to print on.

VL is the memory location that sets volume in sound generation. By itself, it can make little clicks.

HD(I) where I = 0 to 7 is the Ith digit of the hex numbers produced, from 0 to 15.

H\$(0 to 15) are the individual numbers in hex, from 0 to F.

I is a FOR-NEXT loop index used locally.

C\$ is a one-key option response from the user.

H\$ is the hexadecimal number INPUT from the user as a string.

L is the length of the string representations of the input numbers.

E\$ is a string of nine blank spaces used to blank out old prompts.

D = 0 to 9 is the value of each digit of input hexadecimal numbers.

D\$ is each character of H\$ in turn; D is the hex value of each D\$.

DC is the decimal sum of each hexadecimal digit.

T\$ is the entire decimal number INPUT to be converted to hex.

M\$ is each character of T\$ in turn, always a one-character string.

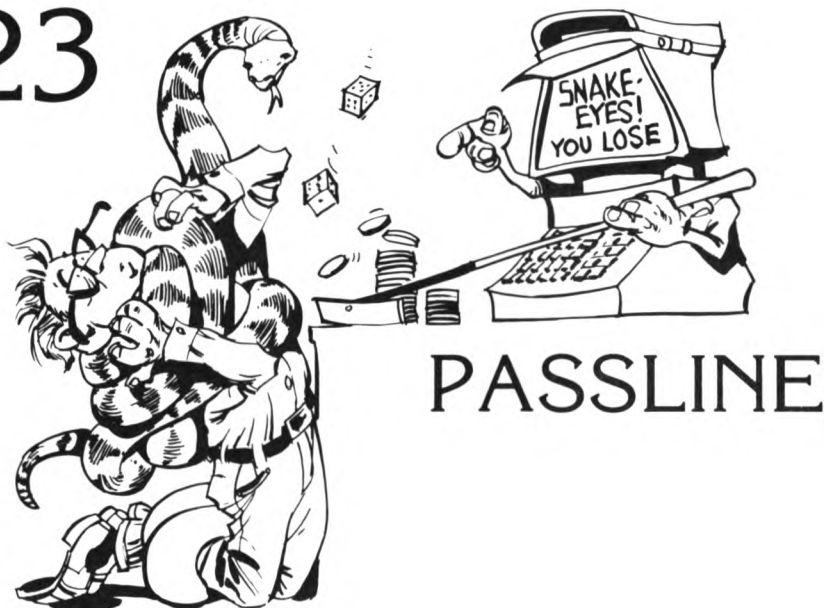
Z9 is a flag set equal to 9 if an input character is "X." X is the cue to go to the other branch of the program.

P = 7 to 0, the successively decremented powers of 16.

S equals 16 to the Pth power to see if it can be subtracted from T.

T is the rounded down numerical value of T\$ (T-string).

23



Fibonacci numbers are the series 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 . . . where each number is the sum of the preceding two numbers. They are named after an Italian mathematician Leonardo Fibonacci da Pisa who was counting how fast rabbits multiply if they don't die of old age. This, plus his advocacy of Arabic numbers over Roman numerals, among other things, is contained in his book *Liber Abaci*, published in 1202 and 1228.

Counting rabbits may seem idle; Fibonacci mentioned his number series seemingly in passing. To this day, here and there in number theory and in nature, we find the series in the most unrelated places (for example, in the number of spirals in the arrangement of seeds in the sunflower). There is an ongoing unofficial competition for the sunflower with the highest exact Fibonacci number! There exists, I hear, a Fibonacci Number Society that trades new and interesting discoveries relating to the series.

I discovered (not originally, of course) an application for the number series at a Nevada crap table. This is where the dice game "craps" is played. There is nothing improper about the name, despite its muddy sound. Most likely, the slang term derives from the gambling environment where "crap dice" is a pass where you lose your entire stake on the first throw of the dice.

The venerable American crap game has different forms. These are best exemplified by the casino version played in Nevada. The most popular bet is the passline. Here you either double or lose your wager. On a per-

centage basis, this is just about the least costly ordinary bet in the casino. The house advantage, by the odds, is roughly 1.4%, a fraction of the usual average take for other games. It is this nearly coin-toss fair play that attracts large bettors; it also attracts system players. The game Passline, as shown here, is nothing if it is not a progressive betting system.

After a gambler puts his wager onto a passline on the crap table, someone shoots the dice. The first roll is very important. If the two dice add up to seven or eleven, all passline bettors win. If two, three, or twelve come up, he loses. That is a "crap roll." Any other number is considered a "point." Once this point is established, shooting will continue until a seven is rolled or the point is "made," meaning it comes up again. If the point is made, the passline bettor wins. If seven comes up first, he loses. The passline player is betting that the shooter will win the next pass or decision.

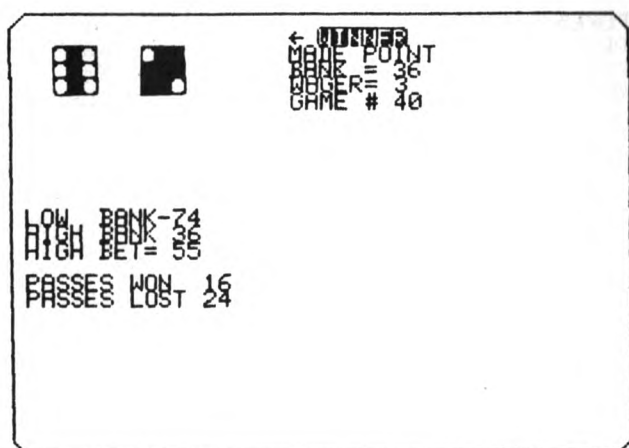
Using the calculated odds, the player should win about 49% of the time and lose the other 51%. The system consists of betting Fibonacci numbers of dollars. When you lose, your next bet is the next higher F number. When you win, you reduce your next wager to the next lower F number. The flaw in such a simple method is that, sooner or later, that house advantage will creep up on you. Slowly, or perhaps quickly, you will be betting F numbers that are too large for comfort.

The saving grace lies in the fact that winning any one bet pays for the preceding two lost wagers. That is the Fibonacci series. This means that every time you win, you can reduce your F number bet by two places in the series, while losing only escalates matters by one place. You will wind up playing for one or only a few chips all night, and that may get boring.

Splitting the difference, why not retreat two F numbers upon winning half of the time and back up only one place on the other alternate wins? You still only need to win an average of 42% of the time, and the odds favor your winning 49% of the time! Add to this the fact that you tend to win when the stakes are the highest and lose when the stakes are the lowest.

The program that follows strictly obeys the rules of craps and the requirements of this system. It is much better to let a machine impartially test any such gambling method. People have a tendency to give themselves the benefit of the doubt where there is a lapse of memory. Casinos don't permit lapses of memory (or funds).

Finally, don't consider this system as a source of easy money. In order to ensure that the program "wins," it is often forced to bet scandalous amounts, far in excess of any reasonable casino table limits. Even if the casino allowed such bets, most people would lack the resources, not to mention the fortitude, to lay wagers of thousands, tens of thousands, and so on. Perhaps the reader can improve upon the algorithm.



LIST #23: PASSLINE

```

1 REM*****
2 REM* PASSLINE          COMMODORE 64 *
3 REM* BY LARRY HATCH    (C) 1983 *
4 REM* MENLO PARK, CALIF    V11.23 *
5 REM*****
100 CLR:POKE53281,0:PRINT"PASSLINE"
110 DIM O$(4),D$(6,3),F(100),RD,DN:T1=TI
120 QA=198:QB=214:VL=54296:P(1)=54273:P(2)=54280
130 P3=54287:A(1)=54277:A(2)=54284:W(1)=54276

```



```

140 POKE53280,11:W(2)=54283:F(0)=1:F(1)=1:F(2)=2
150 PRINT"THE FIBONACCI NUMBER SERIES WAGERS:"
160 FORI=3TO43:F(I)=F(I-1)+F(I-2):PRINTF(I),
170 NEXTI:REM* CREATION OF THE DICE FOLLOWS
180 O$(0)="  " :O$(1)="  " :O$(2)="  "
190 O$(3)="  " :O$(4)="  "
200 FORI=1TO3:D$(0,I)="  ":NEXTI
210 D$(1,1)=O$(0):D$(1,2)=O$(2):D$(1,3)=O$(0)
220 D$(2,1)=O$(1):D$(2,2)=O$(0):D$(2,3)=O$(3)
230 D$(3,1)=O$(3):D$(3,2)=O$(2):D$(3,3)=O$(1)
240 D$(4,1)=O$(4):D$(4,2)=O$(0):D$(4,3)=O$(4)
250 D$(5,1)=O$(4):D$(5,2)=O$(2):D$(5,3)=O$(4)
260 D$(6,1)=O$(4):D$(6,2)=O$(4):D$(6,3)=O$(4)
270 POKEQB,20:PRINT
280 POKEQA,0:PRINT"READY? ♦":WAITQA,1:POKEQA,0
290 PRINT"♣":T2=TI-T1:T1=T2:SD=-T2/113:X=RND(SD)
300 FI=1:G=1:BL=0:BH=0:WH=0
310 T2=TI-T1:T1=T2:SD=-T2/11317:X=RND(SD)
320 W=F(I):PRINT"♠"TAB(18)"WAGER="W"
330 R=0:PRINTTAB(18)"GAME # "G"
340 IFW>7.2E8THENPRINT"♠YOU'RE BUSTED!":END
350 P=0:PRINT"♠"TAB(18)"NEXT PASS" :GOSUB690
360 IFP=0AND(DT=7ORDT=11)THENR=1:GOSUB440:GOTO320
370 IFP=0AND(DT=20ORDT=30ORDT=12)THENR=-1:GOSUB440
380 IFR<>0THEN320
390 P=DT:REM*POINT ESTABLISHED
400 PRINT"♠"TAB(18)"POINT="P"
410 GOSUB690:IFDT=7THENR=-1:GOSUB440:GOTO320
420 IFDT=PTHENR=1:GOSUB440:GOTO320
430 GOTO400
440 IFR>0THEN500:REM* WIN/LOSE SUBROUTINE
450 R$="♠" :B=B-W:FI=FI+1:LS=LS+1
460 IF DT=7THENPRINT"♠"TAB(18)"SEVEN OUT"
470 IFDT<40ORDT=12THENPRINT"♠"TAB(18)"CRAP DICE"
480 POKEW(1),0:POKEVL,15:POKEP3,12+DT+5
490 POKEA(1),12:POKES(1),28:POKEW(1),19
500 IFR<0THEN 580
510 R$="♠" :B=B+W:FI=FI-1:WN=WN+1
520 IF FI<1 THEN FI=1
530 IF DT=PTHEN PRINT"♠"TAB(18)"MADE POINT"
540 IFDT=7ORDT=11THENPRINT"♠"TAB(18)"A NATURAL"
550 POKEW(1),0:POKEVL,15:POKEP3,DT+9
560 POKEA(1),12:POKES(1),28:POKEW(1),21
570 IF WN/2=INT(WN/2)THENFI=FI-1:IFFI<1THENFI=1
580 PRINT"♠"TAB(18)R$
590 PRINT:PRINTTAB(18)"BANK ="B" :G=G+1
600 IFB<BLTHENBL=B
610 IFB>BHTHENBH=B
620 POKEQB,10:PRINT:IFW>WHTHENWH=W
630 PRINT"LOW BANK"BL"
640 PRINT"HIGH BANK"BH"
650 PRINT"HIGH BET="WH"
660 POKEQB,14:PRINT:PRINT"PASSES WON "WN"
670 PRINT"PASSES LOST"LS"

```

```

680 FORZ=20T080:POKEP(1),Z:NEXTZ:RETURN
690 POKEVL,12:REM*ONE TWO-DIE TOSS
700 DT=0:FOR DC=1T02:GOSUB760:POKEW(DC),0
710 DT=DT+RD:POKEA(DC),41:POKEP(DC),(9+3*RD)*DC
720 PRINT"¤":FORDL=1T03:PRINTTAB(6*DC-4)D$(DN,DL)
730 NEXTDL:POKEW(DC),49-16*DC
740 NEXTDC:PRINT"¤"TAB(18)"ROLLED"DT"¤"
750 FORZ=1T0100:NEXTZ:RETURN
760 RD=INT(30*RND(1)+1+DN):REM SINGLE DIE TOSS
770 IFRD<7THENDN=RD:RETURN
780 RD=RD-6:GOTO770

```

LINE ANALYSIS FOR PASSLINE

100-110 Clear variables; make screen field black; PRINT title in white; DIMension arrays and variables; note the time in jiffies.

120-140 Assign keyboard, print-line, and sound-effect memory location constants; turn screen border brown; assign first elements of the Fibonacci series array.

150-170 Assign the rest of the F number series up to F(43); PRINT it to the screen.

180-190 Create the four possible slices of any die face.

200-260 Assign the slices above to the two-dimensional string array D\$(i,j).

270-280 Prompt and wait for keystroke to continue.

290-310 Set time-dependent random variables; assign values to all initial variables.

320-410 The main loop of the program. All else is subroutines.

320-340 Make the next wager from the F number series; announce it with the game (pass) number; stop the game if the player loses 720 million dollars or more.

350-380 Announce next pass; throw the dice; branch for 7-11 naturals or 2, 3, 12 crap dice. If a decision is made, go back for the next pass.

390-430 Keep throwing passes until the point is made or a seven comes up first.

440-490 Skip ahead if shooter didn't lose. Otherwise, set R\$ to the outcome message; subtract the wager from the bank B; increment the F-number index FI; increment loss-counter LS; determine cause of loss and print a message why; make a racket signifying a loss.

500-520 Skip ahead if shooter didn't win. Otherwise, announce win-

ner; add wager to the bank; decrement the F-number counter FI; increment the passes-won counter WN. Don't let FI be less than one, our lowest bet index.

530-540 Determine the reason for winning and PRINT it out.

550-570 Make winning noises; decrement the FI counter again on alternate wins. End of winning routine. The losing and winning routines are parts of the larger "pass-ended" subroutine from 440 to 680.

580-610 PRINT out the winner/loser string R\$; PRINT bank statistics; increment pass counter G; adjust high and low bank records as needed.

620-680 PRINT out bank and wager record statistics; make another noise.

690-780 Subroutine; one toss of both dice.

690-710 POKE in sound effects; call single-die subroutine twice; add each to DT = dice total.

720-750 PRINT out the two dice as they land individually; PRINT their numerical total.

760-780 Single-die subroutine; pick a random number up to 30 plus the last die cast RD; calculate $DN = RD \text{ modulo } 6 \text{ plus one} = 1 \text{ to } 6$.

VARIABLE LIST FOR PASSLINE

In order of appearance:

O\$(1-4) are the four possible slices of a die face showing up to two spots each.

D\$(6,3) are the six faces of the die, each having three slices.

F(I) is the Ith Fibonacci number in the number series.

RD is a random die number used to determine DN.

DN is the number that comes up each time one die is cast.

P(1-2) are the locations for voice #1 and voice #2 pitches.

P3 locates the pitch of voice #3.

A(1-2) are attack/decay locations for voices #1 and #2.

W(1-2) are waveform memory locations for the first two voices.

T1, T2, and TI are used for randomization based on time elapsed. TI is an internal variable updated by the machine only.

BL is the lowest player's bank on record during a playing session.

BH is the highest player's bank on record.

WH is the highest wager made yet.

I is an index counter used locally.

FI indexes the Fibonacci series F(FI) to determine the next wager.

G is a game or comeout counter. It advances one for each decision, win or loss.

SD is a dummy argument used to randomize the RND function.

X is a dummy variable to legalize calling the RND function.

W is each wager equaling F(FI).

P is the point or player's object number, should the pass not be decided on the first throw of the dice.

DT is the dice toss, the sum of the faces showing on both dice.

R is -1, 0 or +1, a flag indicating a winner, pass in progress, or a loser.

R\$ declares winner or loser to the screen in reverse field.

WN is how many games or decisions are won by the system.

LS is how many decisions are lost by this gambling system.

Z and ZZ are dummy variables used in time-delay loops.

DC indexes the dice cast from #1 to #2.

DL indexes the three slices of each die face printed to the screen.

RD = DN the total of the two die faces.



ZINGER

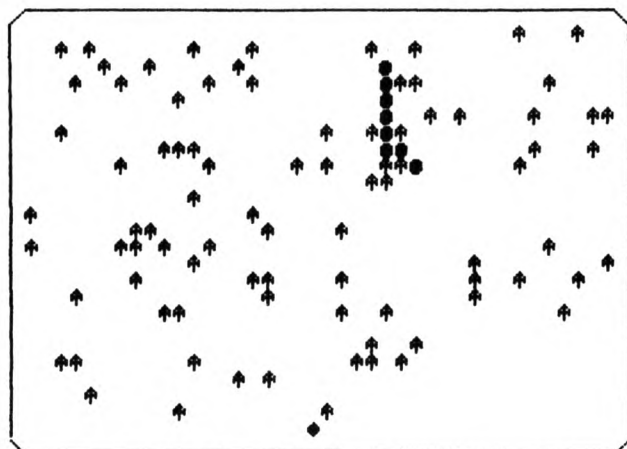
Zinger is inspired by a popular arcade-type game based on centipedes running around a patch of mushrooms. The speed limitations imposed by BASIC leave us with an entirely different game, one that is much simpler but no doubt a lot shorter in length. The Zinger is shown as a series of solid circles that advance rather lazily across the top of the screen. Encountering mushrooms, the Zinger fumbles about until it finds a path around them. The player is shown at screen bottom as a diamond graphic.

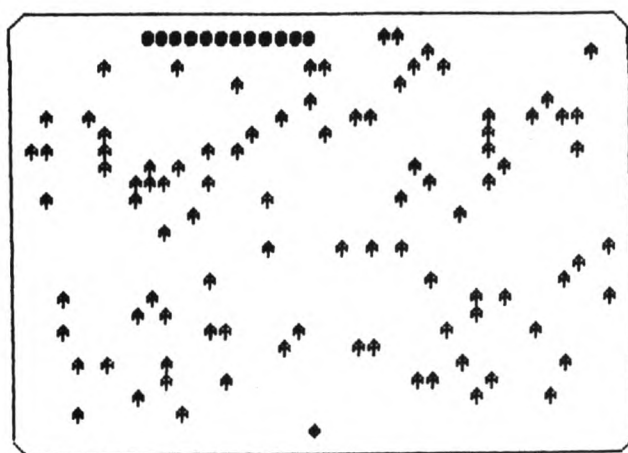
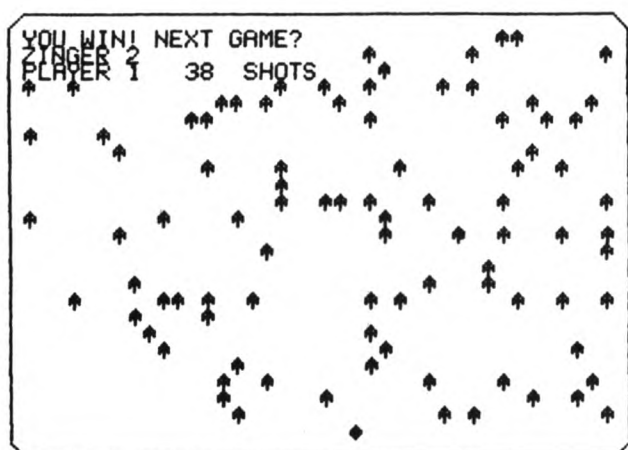
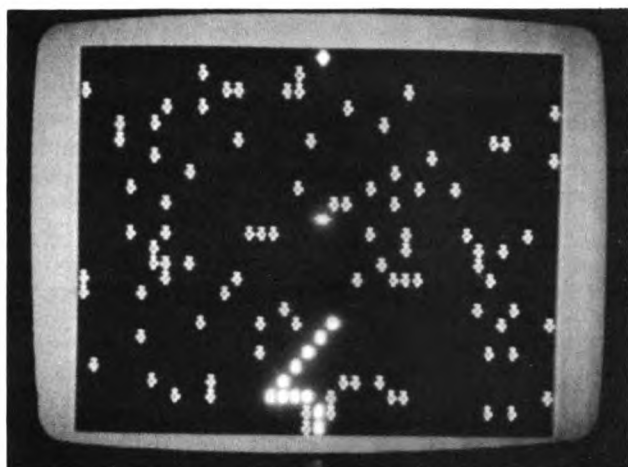
The player uses the left and right cursor keys (ignoring the arrows) to line up the diamond for a good shot. Each time he hits a letter key, he fires a "zap" upward, shown as a blue asterisk. If the asterisk strikes anything, mushroom or Zinger, it disappears along with the zapper.

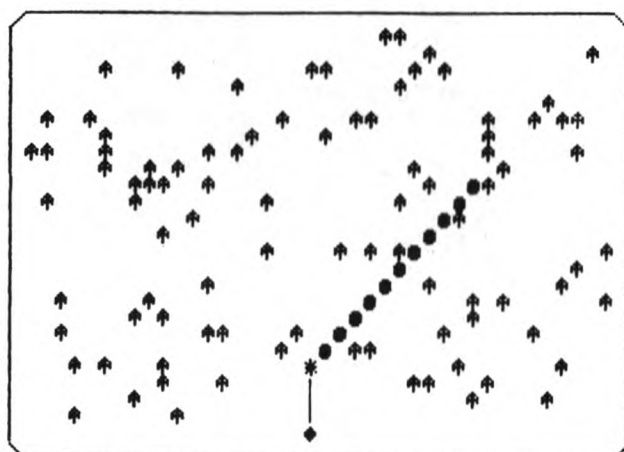
Unlike the arcade game, a bisected Zinger does not fly off into two deadly sections when zapped in the middle. Instead, it pulls itself together in a dignified manner and runs off shorter but in one piece. Each time it is zapped, it loses one section. The shorter it gets, the faster it moves. When it is hit, it will usually pick a random change of direction as an evasive measure. This can be horizontal, vertical, or diagonal, but it is generally downward. If it gets within several spaces of the hero while on the bottom line, it will tear after him.

Since the player can only shoot upward, he has to run using the cursor keys. He can usually get away. Scores are kept for games won and for shots fired. If it becomes easy to win on games, try for the lowest shot score.

Your game count may suffer if you try too hard. You lose the game entirely if the Zinger touches the diamond graphic. When that happens, a zap sound is created and a cross is erected on the spot. This adds a point to the Zinger's score.







LIST #24: ZINGER

```

1 REM*****
2 REM* ZINGER      COMMODORE 64  *
3 REM* BY LARRY HATCH      (C) 1983  *
4 REM* MENLO PARK, CALIF      11.22  *
5 REM*****
100 CLR: DIM C%(13), J(6), HR, ZP, I, IN, FS, LS
110 QA=198: QC=197: SC=1024: CS=55296: VL=54296
120 P1=54273: AD=54277: SR=54278: WF=54276: S=0
130 J(0)=39: J(1)=40: J(2)=41: J(3)=1: J(4)=-1
140 J(5)=-40: J(6)=-39: POKE53281,0
150 PRINT "ZINGER": FS=11: LS=0: IN=1: FOR I=1 TO 99

```



```

160 R=INT(940*RND(1))+20:POKESC+R,88:POKECS+R,8
170 NEXTI:NP=13:HR=979:FORI=LS TO FS:C%(I)=I
180 POKESC+C%(I),81:POKECS+C%(I),3:NEXTI
190 POKESC+HR,90:POKECS+HR,1
200 IFABS(HR-NP)<9 THEN IN=SGN(HR-NP)
210 P=C%(FS):NP=P+IN:R=INT(4*RND(1))
220 IFNP>999THENIN=J(R):NP=INT(39*RND(1)):GOTO240
230 IFNP/40=INT(NP/40)THEN NP=NP-1:IN=J(R)
240 PE=PEEK(SC+NP):IFPE=32ORPE=81 THEN280
250 IF PE=90THEN550
260 R=INT(7*RND(3)):NP=P+J(R):IFNP<20THEN260
270 GOTO240
280 LP=C%(LS):FORI=LS TO FS-1:C%(I)=C%(I+1)
290 IFLS=FSTHEN310
300 POKESC+C%(I),81:POKECS+C%(I),3
310 NEXTI:C%(FS)=NP:POKESC+NP,81:POKESC+LP,32
320 POKECS+NP,3:POKEVL,12:POKEVL,0
330 GETZ$:IFZ$=""THEN190
340 H1=HR:POKEVL,0:POKEVL,12:IFZ$="X"THENHR=HR-1
350 IFZ$="M"THENHR=HR+1
360 IFHR>999THENHR=HR-40
370 IFHR<960THENHR=HR+40
380 POKWF,0:IFHR=H1THEN400
390 POKESC+H1,32:POKESC+HR,90:POKECS+HR,1:GOTO330
400 S=S+1:R=INT(4*RND(3)):POKEVL,8:POKEAD,11
410 POKESR,0:POKEWF,129:POKEP1,60:FORJ=1TO24
420 ZP=SC+HR-40*J:PK=PEEK(ZP):ZC=ZP+54272
430 POKEZP,42:POKEZC,6:IFPK=81THENJ=24:GOTO460
440 IFPK=88THENJ=24:GOTO470
450 IFPK<80THEN480
460 POKESC+C%(LS),32:LS=LS+1:IN=J(R):POKEWF,0
470 POKWF,0:POKEP1,PK-60:POKEAD,7:POKEVL,15
480 POKWF,129:POKEZP,32:NEXTJ:IFLS>FSTHEN510
490 IF PK=32THENPOKEQA,0
500 GOTO190
510 PRINT"YOU WIN! NEXT GAME?":FORI=40TO80STEP2
520 .POKEAD,9:POKEVL,15:POKEP1,I
530 POKWF,33:FORZ=0TO40:NEXTZ:POKEWF,0:NEXTI
540 PS=PS+1:GOTO580
550 ZS=ZS+1:PRINT"YOU LOSE! NEXT GAME?":POKEWF,0
560 FORI=0TO30:POKEVL,15:POKEVL,0:NEXTI
570 POKESC+HR,93:POKESC+HR-40,91:POKECS+HR-40,1
580 PRINT"ZINGER"ZS:PRINT"PLAYER"PS,S" SHOTS"
590 POKEQA,0:WAITQA,1:POKEQA,0:IN=1:GOTO110

```

LINE ANALYSIS FOR ZINGER

100-120 Clear variables; DIMension arrays and simple variables for speed. Assign keyboard, screen, and sound memory location constants for PEEKs and POKEs.

130-140 Assign the jump array for Zinger's surprise moves; turn screen field black.

150-160 Title in white; assign initial values to variables; put 99 mushrooms at random positions on the screen.

170-180 Assign initial screen positions to the Zinger; POKE them in with color.

190-200 POKE the hero into place; Zinger chases the hero if nearby on the bottom line.

210-220 Calculate the next position for the Zinger; keep it on the screen.

230-240 Keep the Zinger from crossing the screen's left edge; allow motion into blank space and back over its own member parts.

250-270 Branch ahead if the hero (player's man) is caught; make random moves to extricate Zinger from mushrooms; jump back for next Zinger motion.

280-320 Advance entire chain of Zinger blobs.

280-290 Hand down position of each blob to the next lower position; jump ahead for a one-blob Zinger.

300-320 POKE in the updated chain of blobs; make front of Zinger equal to the new position NP; color in NP; make a click sound.

330-380 GET and process player's keystroke.

330-340 If no keystroke is made, branch back to Zinger motion routine; record old position of hero; allow for leftward motion.

350-380 Set HR position according to the keystroke; keep the hero onscreen; skip ahead one line if the hero didn't move at all.

390 Moves the hero right or left; goes back for another keystroke if any.

400-410 Shot is fired; increment shot counter; set R for random evasive Zinger motion later; initiate zap sound; start zap-asterisk motion loop.

420-430 Set ZP and ZC character and color locations for the ascending asterisk; POKE them in; branch if the Zinger is hit.

440-450 Test if mushroom is hit and branch; test if nothing hit; branch.

460-480 Shorten the Zinger by one blob; assign his new random direction; make a bang sound; terminate zap loop; branch to win routine if no Zinger left.

490-500 Reset the keystroke counter if nothing was hit; jump back for next Zinger move.

510-540 Player win routine; make a racket; increment player's score; jump ahead for endgame.

550-570 Player loss routine; increment Zinger's score; make a different sound; erect a cross.

580-590 PRINT out all scores; await a keystroke; jump back unconditionally for the next game.

VARIABLE LIST FOR ZINGER

In order of appearance:

C%(I) is the indexed positions of the Zinger on the screen from 1 to 999.

J(1-6) are the six increments needed to move the Zinger head one square in any direction.

HR is the screen position of the hero, the player's man.

ZP is the screen memory location of the zapper as it moves.

I is a local index counter.

IN is the increment chosen, usually from J(R), to change the Zinger's direction.

FS = 11 is the constant Zinger head subscript for C%(I).

LS = 0 to 10 is the Zinger's last index starting point as it gets shorter.

QA is the keystroke counter memory location.

QC is the key-now-pressed register memory location.

SC and CS are beginning memory locations for screen character and color.

P1, VL, AD, SR, and WF are the usual sound-generation memory locations for voice one.

R is a local random variable for various tasks.

NP is the new screen position of the Zinger's head.

P is the previous position of the Zinger's head.

PE is the PEEKed memory contents of NP to determine legal moves.

LP is the screen position of the Zinger's tail end so it can be blanked out as it moves.

Z\$ is any player keystroke as he plays.

H1 is the hero's old position so it can be blanked out.

S is how many shots (zaps) the player fired at the Zinger.

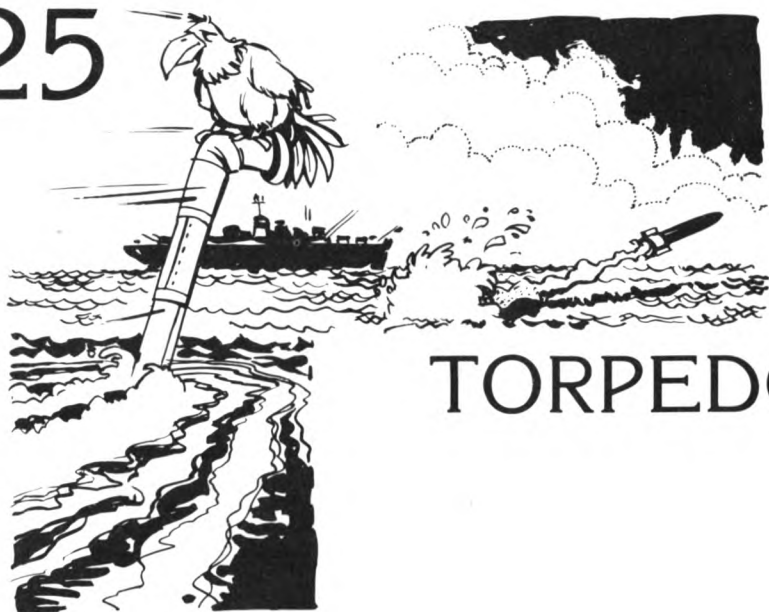
J indexes the zap asterisk up the screen for player potshots.

PK is the contents of screen locations on the zap trajectory.

PS is the player's score in games.

ZS is the Zinger's score in games.

25



In this real-time action game, the player is the skipper of a submarine. It is wartime, and enemy shipping is passing by in the distance on four shipping lanes. From a position at the center of screen bottom, you fire torpedoes in one of nine angles. Having limited speed, unlike a laser gun, you must shoot ahead of the passing craft. Four ships of four random types appear on the sides of the screen and move across it.

The enemy vessels range from a PT boat to a destroyer, each with a different point value when it is torpedoed and sunk. For each battle, you have 20 torpedoes, so shoot carefully. The highest score of any battle so far will be displayed at the end of every battle, and for once the enemy cannot shoot back.

The ships, the torpedo, and the colorful flash are all sprites—six sprites in all. The DATA statements at the end of the program define the images for these sprites and make up nearly a third of the program length. The only PRINT statements used are for instructions and scores, etc., the rest being POKEd into place.

When the program starts, a ship's bell is sounded. There is a short pause as initial variables are set up. Then the bell rings twice while instructions are printed out. The next keystroke will start the action.

The screen is cleared next, and a section of blue sky appears at the

top of the screen to represent the horizon. A double yellow line descends from the top right side of the screen to separate the playing field from the scoring and prompt field. The game would be slowed down considerably if we ran the sprites all the way to the right; lots of extra code would be adjusting the ninth bit of each sprite's x coordinate. One byte, location 53264, has eight bits in it. Each one is the ninth bit of these x coordinates for the eight sprites.

Next, the torpedo sprite appears at the center of the screen bottom, followed by random ships entering from both sides. You fire torpedoes by pressing the 1 through 9 keys; the 5 key fires straight up. Other number keys fire up in angles roughly corresponding to their positions on the keyboard.

Each sprite is serviced (i.e., moved) in what is called a service loop. The torpedo advances, and one by one, the ships move. If this is done fast enough, with small enough increments, the illusion of smooth simultaneous motion is created. BASIC is too slow for that, but I used several tricks to get as close as possible. One trick is to change a repetitive statement like $X = X + 10$ to $X = X + T$, where T was earlier assigned the value of ten. This way T is already stored as ten but in binary (1010 base 2), and BASIC doesn't have to convert from decimal to binary over and over. This makes a significant difference in speed, at the cost of making the LIST more incomprehensible. It also helps to trim the commands in the motion loop to the bone. It does no good to color the same sprite grey over and over, for instance.

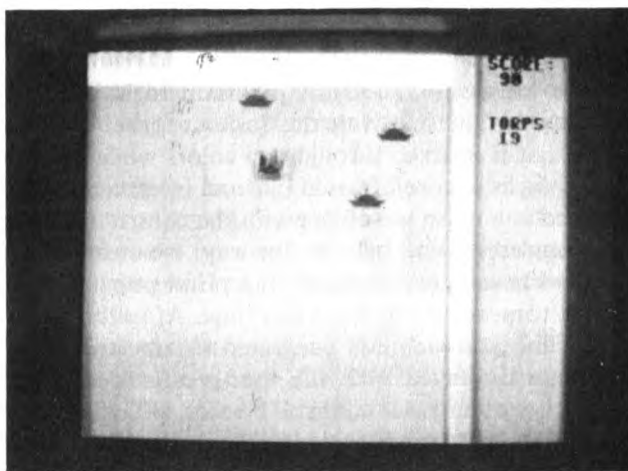
Using attack, decay, sustain, and release, we get a fairly realistic sound of an old torpedo firing with white noise. The attack simulates firing the missile. This decays into the sustain level for most of the distance. The sustained volume level releases to nothing as the torpedo disappears into the blue horizon at the top of the screen display.

When we hit a ship, the sprite collision register (memory location 53278) tells the program to activate the flashout sprite. This is an explosion-shaped sprite that is indexed through ten colors while the sound is generated, and the player's score is raised in small increments.

The sound is a noise waveform with the control bit of the waveform register WF toggled on and off. In this way, we create a nice wrenching explosion sound instead of an anemic cap-pistol pop.

Only one torpedo can be fired at a time. After these are used up, the game is over, and your score is compared to any previous all-time high score. These are displayed with the ever-present decrementing torpedo count. The ships vary in point count. PT boats, which are small, count the most because Admiral Atrocity rides them.

Unless a ship is sunk, it will float offscreen on one side or the other. When this happens, it is assigned a new sprite image (a PT boat can turn into a destroyer), a new speed, and a new direction. It may reappear on the opposite side of the screen. Sometimes the screen will show four of the same type of boat. If four PT boats appear, then Admiral Atrocity brought along his disgusting relatives; 100 points each in any case. New boats are given random speeds and directions also. Which side of the screen they appear on depends upon their direction. At the beginning of a new game, some of the boats will not appear right away. This makes the second and subsequent games more true to life and less of a turkey shoot.



LIST #25: TORPEDO

```

1 REM*****
2 REMTORPEDO          COMMODORE 64 *
3 REMBY LARRY HATCH      (C) 1983 *
4 REMMENLO PARK, CA      V 11.23 *
5 REM*****
100 CLR:QA=198:QB=214:POKE53280,9:POKE53281,5
110 DIM PX(4),SX(4),SY(4),XI(4),S$(4),SH(4)
120 SC=1024:CS=55296:U=54296:P1=54273:P3=54287
130 AD=54277:SR=54278:WF=54276:SN=53269:LC=15872
140 GOSUB750:PRINT"  TORPEDO  "
150 FORI=53248TO53264:POKEI,0:NEXTI:POKE53261,255
160 FORI=0TO383:READX:POKELC+I,X:NEXTI:GOSUB750
170 FORI=1TO4:SH(I)=I:PX(I)=53248+2*I:POKEPX(I),0
180 READS$(I):NEXTI:FORI=0TO6:POKE2040+I,248+I
190 POKE53287+I,11:NEXTI:FC=53292:POKEFC,8
200 CL=53278:P(1)=60:P(2)=50:P(3)=100:P(4)=40
210 POKE53287,0:POKE53275,63:GM=1:HS=0:POKESN,63
220 PRINT"  1  2  3  4  5  6  7  8  9  "
230 PRINT"  YOU ARE FIRING TORPEDOES FROM A SUB."
240 PRINT"  YOUR SHOT COORDINATES ARE SHOWN ABOVE"
250 PRINT"  USE YOUR NAVAL NUMBER KEYS TO ENTER"
260 PRINT"  THESE COORDINATES AND FIRE."
270 FORI=1TO4:POKEPX(I),40:POKEPX(I)+1,105+16*I
280 PRINT"      "S$(I),P(I)" POINTS":NEXTI
290 PRINT"ADMIRAL ATROCITY IS ON THE PT BOAT."
300 PRINT"PRESS ANY KEY TO RAISE THE PERISCOPE."
310 GOSUB750:POKEQA,0:WAITQA,1:POKEQA,0
320 TS=0:TP=20:FORI=53248TO53264:POKEI,0:NEXTI
330 PRINT"  TORPEDO  "
340 PRINT"  "
350 FX=53258:FY=53259:HX=128:HY=240
360 X0=53248:Y0=53249:TY=-3:PRINT"  ";
370 FORI=30TO990STEP40:POKESC+I,160:POKECS+I,7
380 POKEQB,3:PRINT:PRINTTAB(33)"  TORPS:"
390 PRINTTAB(33)TP:REM*
400 POKESC+I+1,160:POKECS+I+1,7:NEXTI:PRINT"  "
410 Z=0:A=1:B=4:TW=2:F=5:C=10:D=18:E=20:W=45
420 FT=50:G=63:K=254:L=255:WP=1.45
430 IFTA%=Z THENXT=HX:YT=HY:GOTO460
440 XT=XT+TX:YT=YT+TY:IFYT<GTHENPOKEWF,128
450 IFYT<FTTHENTA%=Z:POKEQA,Z:GOTO470
460 POKEX0,XT:POKEY0,YT:REM ADVANCE TORPEDO
470 POKECL,0:FORI=ATOB:SX(I)=SX(I)+XI(I):S=SX(I)
480 IFS>KORS<TW THENGOSUB660
490 POKEPX(I),SX(I)
500 NEXTI:P=PEEK(CL):IFTA%=ZTHENPOKEWF,128
510 IF(PANDA)=ZTHEN600
520 PRINT"  "TAB(33)"SCORE:"POKEAD,7:POKEU,15
530 NS=INT(WP*LOG(P-A)):IFNS>4THEN430
540 CX=SX(NS):POKESN,L
550 CY=SY(NS):POKEFX,CX:SI=P(SH(NS))/C:POKEP1,30

```



```

560 POKEFY,CY:FORI=ATOC:POKEFC,I:POKEWF,129
570 TS=TS+SI:PRINT"TAB(33)TS:POKEWF,128
580 POKEFC,10:NEXTI:POKEPX(NS),Z:SX(NS)=0
590 POKECL,Z:POKEQA,Z:TA%=Z:POKEX0,HX:POKEY0,HY
600 POKESN,G:POKEYF,L:IFTA%THEN430
610 IFTP<1THEN700
620 GETF$:V=VAL(F$):IFV=ZTHEN430
630 TP=TP-1:POKEQB,4:PRINT:PRINTTAB(33)TP"II "
640 TX=(V-F)/2:TA%=A:POKEWF,0:POKEAD,61:POKESR,42
650 POKEU,7:POKEP1,60:POKEWF,129:POKEQA,0:GOTO440
660 S%=SGN(2*RND(A)-A):R%=4*RND(A)+1:SH(I)=R%
670 XI(I)=S%*(3*RND(A)+A):POKE2040+I,248+R%
680 SX(I)=TW:IFS%<ZTHENSX(I)=K
690 SY(I)=W+D*I:POKEPX(I)+A,SY(I):RETURN
700 POKEQB,6:PRINT:IFTS>H%THENH%=TS+.1
710 PRINTTAB(33)"RECORD":PRINTTAB(33)H%:GM=GM+1
720 POKEQB,9:PRINT:PRINTTAB(33)"GAME #"
730 PRINTTAB(33):GM"?":POKEQA,0:WAITQA,1:GOSUB750
740 FORZ1=0TO400:NEXTZ1:GOSUB750:POKEQA,0:GOTO320
750 POKEWF,0:POKEU,6:POKEAD,13:POKEP1,87
760 POKEP3,32:POKESR,0:POKEWF,21:RETURN
770 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,16,0
780 DATA 0,56,0,0,56,0,0,56,0,0,56,0,0,56,0
790 DATA 0,16,0,0,56,0,0,108,0,0,0,0,0,0,0
800 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
810 REM** TORPEDO SPRITE ↑↑. SHIPS FOLLOW.
820 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,56,0
830 DATA 4,16,32,2,16,64,17,255,136,11,255
840 DATA 208,7,255,224,255,255,255,127,255,254
850 DATA 63,255,252,31,255,248,0,0,0,0,0,0
860 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
870 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
880 DATA 0,48,0,0,16,0,0,16,0,0,16,0,0,120,0
890 DATA 0,56,0,0,60,0,127,255,254,63,255,252
900 DATA 63,255,252,0,0,0,0,0,0,0,0,0,0,0,0,0
910 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
920 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
930 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
940 DATA 0,60,0,0,126,0,3,255,192,1,255,128
950 DATA 0,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
960 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
970 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
980 DATA 0,24,0,0,24,0,0,24,0,1,255,128
990 DATA 7,255,224,255,255,255,127,255,254
1000 DATA 63,255,252,31,255,248,0,0,0,0,0,0,0
1010 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1020 DATA 0,0,0,2,0,64,3,0,192,3,129,192,3,195
1030 DATA 192,3,231,192,3,255,192,35,255,196,51
1040 DATA 255,204,59,255,220,63,255,252
1050 DATA 63,255,252,63,255,252,63,255,252
1060 DATA 31,255,248,31,255,248,31,255,248
1070 DATA 31,255,248,7,255,224,0,0,0,0,0,0,0
1080 DATA DESTROYER,U-BOAT,PT BOAT,CRUISER

```

LINE ANALYSIS FOR TORPEDO

100-130 Clear variables; turn screen border orange; turn screen field black; assign screen, keyboard, character, color, and sound memory location constants.

140-150 Ship's bell sounds; title in white; POKE sprites all off; put sprite #6 away from the others.

160-190 Ring second bell; READ and POKE sprite images from DATA into user memory above location 15827; assign ship's x-position array; set them offscreen; READ in the names of ship's types; point each sprite to its image; color each sprite; assign flash-sprite color location FC; color it.

200-210 Assign scores for hitting ship types; color torpedo black; give sprites display priority; turn first six sprites on.

220-310 PRINT instructions; ring third ship's bell; wait for key-stroke to continue.

320-340 Reset scores; put torpedo and sprites offscreen; clear screen and retitle; put in two lines of blue sky.

350-360 Assign fireball and torpedo X, Y control memory locations; assign X, Y position coordinates of torpedo; TY is upward torpedo Y increment; send cursor home.

370-400 Fill in yellow border to display between fields; put in TORPS prompt.

410-420 Assign common constants for algorithm speed.

430-460 Test if torpedo is active; branch ahead if not; increment torpedo X, Y coordinates; torpedo disappears offscreen if it goes too high; POKE torpedo into its new coordinates.

470-490 Clear sprite collision register; advance ships along x axis; branch for a new ship if one goes offscreen; POKE Ith ship into its new place.

500-510 PEEK into collision register CL; turn sound off if no torpedo is moving; branch to line 600 if no collision with a ship.

520-550 Set explosion sound values; determine number of the ship struck; determine coordinates CX,CY of the ship struck; set flash sprite at CX,CY.

560-590 Rotate the flash colors while making explosion sound and incrementing player's score; final flash color set; flash sprite disappears; torpedo made inactive and sent home.

600-620 Turn regular sprites on; move flash sprite offscreen; branch ahead if no torpedoes left to shoot; GET player's keystroke and branch back if invalid.

630-650 Fire the TORP routine; decrement torpedo counter; display the count; calculate new torpedo X-position increment TX; set torpedo-active flag; set ADSR sound parameters for firing; start the sound; jump back into sprite motion service loop.

660-690 Subroutine; create a new ship; random direction, speed, image, and origin all assigned; set into the correct shipping lane.

700-740 Out of torpedoes endgame; update the high score; PRINT it out; increment game counter; wait for a keystroke; ring ship's bell twice; go back for next game.

750-760 Subroutine; sound the ship's bell.

770-810 Torpedo sprite image in DATA.

820-860 Destroyer sprite image.

870-910 Submarine sprite image.

920-950 PT boat sprite image.

960-1010 Cruiser sprite image.

1020-1070 Explosion-flash sprite image.

1080 DATA for ship's type-names.

VARIABLE LIST FOR TORPEDO

In order of appearance:

QA and QB count keystrokes and position the line cursor for PRINTing.

PX(n) is memory location controlling nth ship's X coordinate.

SX(n) is nth ship's X coordinate, POKEd into PX(n).

SY(n) is nth ship's Y coordinate.

XI(n) is nth ship's X-position increment.

S\$(n) is name of nth type of ship.

SH(n) is sprite image number of the nth ship for a new image.

SC, CS is beginning locations in screen character and color memory.

U is volume of sound generator, usually designated VL, a memory location.

P1, P3 are memory locations controlling frequencies of voices #1 and #3.

AD, SR, WF are attack/decay, sustain/release and waveform of voice #1.

SN = 53269 is the sprite control register (which sprites are on or off).

LC = 15287 is the beginning location of sprite images as we POKE them in.

I and Z1 are common index counters in FOR-NEXT loops used locally.

FC = 53292 is the color control location for sixth (flash) sprite.

CL = 53278 is the sprite collision register.

P(n) = 40 to 100 is points for sinking the nth ship.

GM is the game (battle) counter (increments every 20 torpedoes).

HS is the highest score yet attained.

TS is the present score.

TP is how many torpedoes are left out of the original 20.

FX, FY are memory locations controlling X, Y coordinates of the flash sprite.

HX, HY are flash sprites' X, Y coordinates POKEd into FX, FY.

X0, Y0 are memory locations for X, Y coordinates of sprite zero (torpedo).

TX, TY are increments to add to torpedo X, Y coordinates for motion.

XT, YT are present torpedo coordinates.

Z, A, B, TW, F, C, D, E, W, FT, G, K, L, and WP are constants set ahead of time to speed up algorithms.

TA% is a flag set to indicate active torpedo in motion.

P is the contents of the collision register.

S is the temporary X coordinate of the Ith ship to see if it ran off-screen.

NS is the number of the ship (1-4) struck by a torpedo.

CX, CY are coordinates of the ship struck to move in the flash sprite.

SI is the score increment, so the player's score goes up in little parts instead of all at once.

F\$ is the player's keystroke (1-9) to fire a torpedo at an angle.

V tests if F\$ is in range or valid.

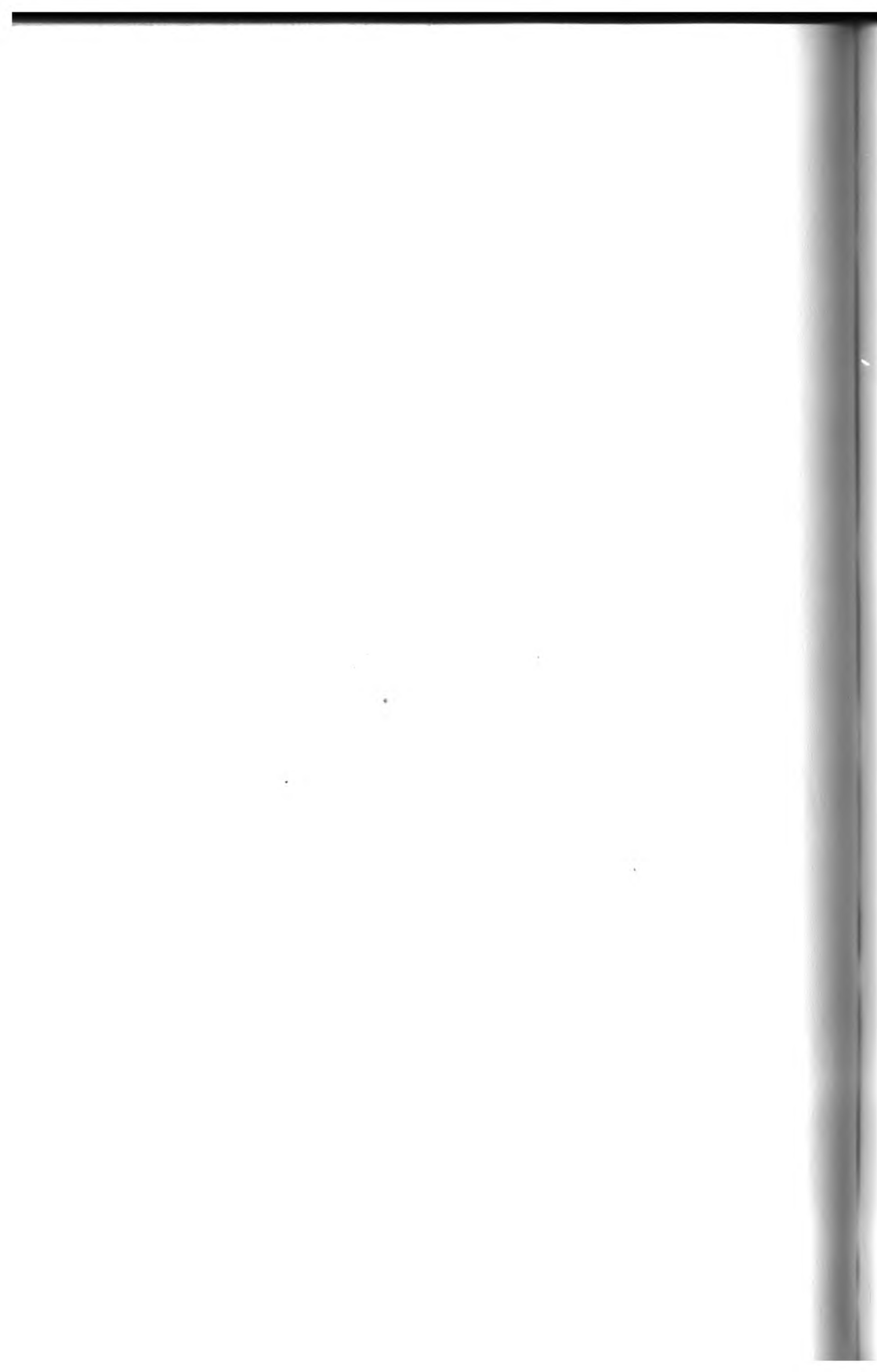
S% plus or minus one determines direction of a new ship.

R% = 1 to 4 is the type of ship the new target will be randomly.

H% is the highest point score so far for any game (battle).



APPENDIX



PEEK and POKE memory locations

The numbers that follow are the addresses of some useful locations in memory. This is a sketchy list intended only to help understand the game programs. No doubt there are many others. For more detail, the author recommends pages 311–334 and other information from the *Programmer's Reference Guide* published by Commodore with Howard Sams Inc.

0: The data direction register. This is where NOT to POKE. If you POKE into a location L and forget to assign L an address number, then you POKE into location zero. Most likely, the cassette recorder motor will turn on, giving you a clue what went wrong.

1: The microprocessor input/output register. Don't even think of POKE-ing anything here.

197 (QC): This register, constantly updated, holds a different number for any key being held down right now. If no key is being pressed, the contents of location 197 is 64.

198 (QA): The keystroke counter. Every keystroke increments the contents of this counter, so BASIC knows how many characters to get when you press return. At that time, it is reset to zero. We can POKE it to zero and WAIT for it to increment to detect or count keystrokes.

214 (QB): This register controls which of 25 lines the screen will print onto next. POKE 214,20:PRINT:PRINT"HELLO" will print "HELLO" on the

21st line. The extra PRINT statement in the middle is required, and this unexplained requirement hasn't changed since the earliest Commodore PET computers.

646: Control of color for PRINT statements. If you POKE 646,1 in the middle of a bunch of lines of PRINT statements, all subsequent text will be printed in white regardless of the original color.

820-827: Unused! A good place to store game scores where BASIC cannot alter them. Only turning off the machine will change the contents.

829-1019: The buffer memory area used to hold blocks of bytes to or from the cassette recorder. A useful place for short machine-code routines, sprite images, etc. Use caution; the lower 64 bytes appear to be used unannounced by other BASIC routines that also find it handy!

1020-1023: Unused and open.

1024-2023: Screen character memory, organized 40 characters by 25 lines.

2024-2039: Unused.

2040-2047: Eight bytes. Multiply the contents of each byte by 64 and you have the starting address of the nth sprite's image in memory, from sprite zero to sprite seven.

2048-40959: The usual area reserved for BASIC programs with all their strings, variables, DATA, etc.

53248-54271: The VIC video chip; color, sprites, high resolution, etc.

53248-53264: Control of X, Y coordinates for each of the eight sprites.

53264: Each bit in the contents of 53264 is the ninth bit of the X coordinates of the 8 sprites. This permits sprites to go more than 255 pixels toward the right side of the screen.

53265: A VIC chip control register. Bit 5 is interesting; if you set it to one, the machine goes into dot-resolution mode. See *Fill the Curve*, page 167.

53269: Sprite on/off register. Each bit turns a sprite on or off for one or zero.

53275: Sprite/background display priority. Each bit gives one sprite the ability/disability to eclipse other characters, etc., on the screen.

53280: Control of border color around the active screen field. Any of 16 colors and shades. Set for attractiveness and minimum eyestrain.

53281: Control of the field background color. Any of 16 shades. Set for maximum legibility and clarity. Choose character colors that contrast; some are barely distinguishable. The C-64 wakes up in shades of blue that are next to illegible.

53287-53294: Color control for each of the eight sprites.

54272-55295: The sound synthesizer chip.

54272, 54273: Voice #1 pitch (frequency) fine, coarse setpoint.

54276: Voice #1 waveform and control register.

54277: Voice #1 attack/decay parameters, four bits each.

54278: Voice #1 sustain/release parameters, four bits each. For voice #2, simply add seven to the register addresses of voice #1. For voice #3, add 14 to the voice #1 addresses.

54293-54296: Addresses of various filter controls. Sound filters weren't used in this book. The author prefers to let the sound chip squawk away.

54296: (VL) The volume of the entire sound chip with all three voices. Only the lowest four bits control sound, so POKE VL, 0 to 15. The four "high" bits help control filters.

55296-55395: The 1000-byte color memory for the screen characters. This corresponds to the 1000 byte character memory from location 1024 to 2023. Numbers must usually be POKEd into both character and color memory for a character to appear on screen.

Most sound and color POKEs are divided into bit slices of four bits each. In some cases, only four bits are present to record a number POKEd there, usually color memories. In other cases, as in attack/decay, there are two four-bit registers sharing one eight-bit byte! This requires us to multiply attack (0-15) by 256 before adding it to decay (0-15). Then we take the sum and POKE that into location AD (54277 for voice #1). With 64k of memory, and three decks of bank-switched devices sharing much of the same areas of the memory map, I find it hard to understand why these important registers were not given different addresses.

Nevertheless, I think the SID sound synthesizer chip is the most impressive innovation in the Commodore 64. As these programs were written, nothing was more fun to work with than the sound.



GLOSSARY

ADDRESS: Any number that uniquely defines a location in memory.

ALGORITHM: A series of logical or mathematical steps, often involving repetition, that calculate or perform a special task.

ANALOG: As opposed to digital; any continuously variable entity, such as a voltage, pressure, or current may be viewed as analog information.

ARGUMENT: In TAN(A), RND(A), SQR(A), and other such functions, A is the argument of the function.

ARRAY: Any subscripted variable in basic, e.g., V(I,J), which is subscripted in two dimensions.

ASCII CHARACTER: Any character as defined by the ASCII Code which arbitrarily assigns code numbers to all such characters.

ASCII CODE: A number coding system for letters, numbers and other symbols.

ASSEMBLER: A program that takes assembly language mnemonics like LDA, STA, or JMP and translates them directly to machine language.

ATTACK: In sound synthesis, the initial rise time of the volume of a sound when it first is heard. A gunshot has an attack time of zero, exploding instantly to full volume.

BASE (n): The number of digits (n) in a numbering system. Decimal numbers are base 10. Hexadecimal is base 16 and binary is base two.

BIT: The smallest amount of information that can be transmitted greater than no information. The quantum unit of data. A light that is either on or off, and which may be described in no other way, can transmit one bit of information at a time.

BIT MAPPING: Assignment to memory of every bit (pixel) on the CRT screen so they can be individually turned on and off for high-resolution graphics.

BOOLEAN ALGEBRA: After George Boole, the algebra of logical operators like IF, AND, OR, NOT, THEN, NAND. Originally a means of deducing logical conclusions with a maze of input.

BRANCH: To jump to another part of a program when or if some conditions are met. For example: IF X=1 GOTO 510. Assembly language has several branch instructions, and they work in the same way.

BUFFER: (1) A portion of memory used to temporarily store input or output between a computer and a peripheral such as the cassette buffer memory space; (2) an integrated circuit that protects delicate components from harm by providing electrical isolation from the outside world.

BUS: A group of wires or conductors that transmit data 4, 8, 16, 32 . . . bits at a time.

BUST: From Blackjack game; to lose because a hand is over 21 points.

BYTE: In the world of small computers, a BYTE is eight bits of information.

CHARACTER: Any letter, number, graphic, or other elementary symbol.

COBOL: A business-oriented language for the mainframe computers. COBOL is usually compiled.

COLUMN: On the C-64 video screen, a column is a vertical row of 25 character spaces.

COMMAND: A direct human or machine instruction to a computer that is not part of a program being run.

COMMAND MODE: The condition in which the C-64 will accept any keyboard commands including alterations to programs. The READY prompt and flashing cursor indicate a return to the command mode.

COMMODORE ASCII: See PET ASCII.

COMPILER: A program that translates programs in Fortran, COBOL, and other high-level languages (like BASIC) into machine language. One instruction in BASIC could represent a whole machine language subroutine.

CONCATENATE: To hook together without spaces, like adding strings.

CONSTANT: A number or algebraic letter which stands for a number that does not change value.

COORDINATES: In these games, the X and Y numbers, in pairs, that determine the position of a pixel, sprite, or character on the two-dimensional screen. A three-dimensional space would require three coordinates, X, Y, and Z.

CRASH: Any breakdown in the normal operation of a computer or peripheral. These often result in endless loops.

CURSOR: A visible or invisible place on the screen where the next printed character will appear. The cursor is visibly blinking when the Commodore 64 is in command mode and invisible while executing programs.

DECAY: In sound synthesis, the time taken for a sound to diminish in volume to its sustained volume level, which is constant.

DECISION (computer): The outcome of logical steps in a computer program making some choice.

DECISION (gambling): Any outcome of a game that decides the ownership of a wager.

DIGIT: (1) Each of the n symbols in a number system of base n ; (2) the place, to the left or right of the decimal point, into which numbers are set in powers of n .

DIGITAL: As opposed to analog; transmitting or holding information in the form of discrete, discontinuous numbers, typically ones and zeroes.

DIMENSIONS: From geometry, directions at right angles to each other. The array $S(I,J)$ is subscripted in two dimensions, I and J . You may think of this as a sheet of numbers I long by J wide.

DONGLE: An electrical device plugged into a computer port as an interface device or to permit use of protected software.

DUMMY VARIABLE: A variable whose value is of no concern. It is placed into a function only because the function demands some variable.

EPROM: Erasable (using ultraviolet light) Programmable ROM.

EXECUTE: To perform a rigidly specified set of instructions such as those found in a computer program.

FILTER: Any device used in sound synthesis to attenuate (tone down) high, low, or any specified frequencies for special effects.

FOR: In BASIC, the symbol indicating the start of an indexed loop.

FORTRAN: A widely used compiled language for mainframes. Difficult to write, it offers the user absolutely amazing speed of execution.

FUNCTION: A mathematical relation defining a unique value to a dependent variable (say Y) for every value of the independent variable (say X).

GET: In C-64 BASIC, the GET statement will literally GET one character from the keyboard or other designated device with no formalities. If no information is present, you will GET a null string or the number zero.

GLITCH: Any real or imaginary electrical surge capable of causing a computer crash or loss of data. The homely glitch (it does exist after all) is accused of every evil known to electronics and computing circles.

HEXADECIMAL (hex): Pertaining to a system of 16 digits from 0 to F. It is a convenient way for people to express long binary numbers since they are easily converted. 10000 in binary equals 10 in hex (16 in decimal).

I-LOOP (or any-other-letter-LOOP): A FOR-NEXT type loop indexed by I or any-other-letter.

INCREMENT: To add to. To raise in value by a small amount; also called the Increment, often one.

INDEX: A number that counts, often in integers, so that a loop and all the indexed variables in it have constantly changing values. Indexes (indices) usually count up or down in constant increments, sometimes set by the STEP condition in a FOR-STEP-NEXT routine.

INPUT: A basic statement with a routine all its own. It is used to get data from the user of a program.

Input: Any information that reaches a computer from the outside world.

INTERPRETER: An operating system that accompanies interpreter languages like BASIC. In the C-64, the interpreter is stored in ROM. It takes every basic statement in your program and analyzes it on the spot for calling machine code routines while you are running. Contrast this to a compiler, which translates source into object code. From then on, you execute only the object or machine code compiled.

J-LOOP: *See* I-Loop.

JUMP: In assembly language, JMP is a mnemonic for an unconditional jump to some address where presumably another executable machine instruction exists. In BASIC, a jump is performed by the GOTO statement, which sends you to a new line number in the program.

LANE: For the purposes of certain game programs, a straight consecutive row of places on a game matrix needed to win or lose.

LEAST SIGNIFICANT DIGIT: The digit in a number with the lowest power of the base. In binary, the least significant bit.

LOCATION: With reference to the screen, a memory location into which numbers may be POKed to make characters appear.

LOOP: A part of a program that jumps back to its beginning, thus repeating over and over. An endless loop is usually the result of some programming error.

MACHINE LANGUAGE: A series of binary numbers that stand for instructions and data which is directly executable by the microprocessor. For convenience, these numbers are expressed in Hex by programmers.

MAIN LOOP or routine: The set of instructions performing the major repetitive routine of a program. Other parts of the program are preliminaries, ending routines, and subroutines called by this main loop.

MATRIX: Any orderly arrangement of numbers or symbols into rows, sheets or cubes, from one dimension on up to hypercubes of n dimensions.

MNEMONICS: In assembly language, the three-letter names given to the machine instructions. For example, TAX, LDA, STX, RTS, NOP . . .

MOST SIGNIFICANT FIGURE: The digit in a number having the highest power of the base; the left-most digit.

OBJECT CODE: The machine language program that is produced when a compiler translates from a source code program, perhaps written in Fortran, COBOL, or even BASIC.

OP-CODE: A binary number (often expressed in hexadecimal) which the microprocessor will see as a specific instruction.

OPEN: A basic statement to open a file. This is how the C-64 communicates with its peripherals. Remember to always close files when you are done.

OPERAND: Any number treated by an operator or function, whether a dependent or an independent variable.

OPERATOR: Mathematically, any math function that converts one operand into another. Basic math operators include TAN(x), RND(x), EXP(x), SQR(x), and many others.

PARITY BIT: In standard (not Commodore) ASCII and other information transmitting codes, an extra bit of information set aside which depends on the sum of all the others. This will usually indicate lost or garbled bits to call for error routines or retransmission of data.

PET ASCII: Similar to standard ASCII but with no parity bit. The extra bit is used to generate an alternate set of characters, including graphics and reverse field. The C-64 uses PET ASCII.

PIXEL: The smallest individually displayable image on the screen; one dot. Commodore machines (after the Vic-20) can individually program these dots on the screen. This is called *dot resolution graphics*.

POSITION: In these games, screen position, from 0 to 999 stands for the 1000 places on the screen that a character can be displayed. Contrast with screen location from 1024 to 2023.

PROM: Programmable Read Only Memory. PROMs are usually programmed by electrically blowing tiny fuses inside. Once programmed this way, their contents are inalterable.

PROMPT: A brief message to the screen indicating readiness for some user action or input from the keyboard. A cue to the user.

RAM: Random Access or Read-Write Memory. You can store numbers in RAM for retrieval later. The memory contents of RAM are lost when power is turned off.

REAL VARIABLE: As opposed to integer or string variables. A standard number with decimal fractions and an exponent when necessary. The name comes from Fortran, which also handles imaginary or complex variables.

RELEASE: The time taken for the volume of a synthesized sound to diminish to nothing after the sustaining period.

ROM: Read Only Memory. Mass-produced memory chips whose memory contents were masked into them at the time of manufacture.

ROUTINE: A set of instructions, coherent and useful, that will be called routinely for execution. A particularly common series of tasks for the computer to perform.

SCREEN: The cathode-ray tube used for visual display. The display itself.

SCROLL: For the entire display to roll up one line, permitting the computer to print on the new blank bottom line. The top line of data is lost to the display.

SOURCE CODE: In compiler languages, the English-like list of statements that the programmer writes and stores. The compiler translates the source code into object code that the machine can understand.

SPRITE: In the C-64, a 24 by 23 pixel character, dot programmable, that can be shown on the screen independent of POKed and PRINTed characters.

STACK: In the 65xx microprocessors, the last-in, first-out register that keeps track of machine addresses for FOR-NEXT loops and subroutines.

STATEMENT: An instruction to the computer, contained in a source or interpreter language program. In the BASIC interpreter, the statements are interpreted and performed directly from the source language.

STEP: The step condition in a FOR statement sets the increment added to the index on each new pass through the FOR-NEXT loop. If unspecified, we step by one at a time.

STRING: Any series of symbols that the computer can manipulate and display. Not numbers; no mathematics can be done on strings.

SUBROUTINE: For our purposes, a subroutine is any routine called with a GOSUB statement and ending with the RETURN statement.

SUSTAIN: In sound synthesis, the volume (not time) at which a tone is maintained after attack and decay, which are usually louder.

TOGGLE: To alternate between logical states or numerical values.

TOKEN: A one-character symbol used by the C-64 to stand for a statement in source code. The question mark is a token for the PRINT statement. Most basic statements are stored as tokens to save memory.

VARIABLE: A number whose value may change at any time. For this reason it is given a letter name like X or Y.

WORD: The word of the C-64 is the 8-bit byte. In larger systems 16-, 32-, and 64-bit words are not uncommon. It is the usual amount of information transmitted via the data bus in any given clock cycle.

X COORDINATE: For graphics displays, the X coordinate refers to one of 40 columns or 320 pixels on the screen from left to right.

Y COORDINATE: The Y coordinate refers to one of 25 lines or 200 pixels on the screen from the top down. An X and Y coordinate define a unique screen position. If we let $N = X + 40 Y$, we can POKE a character to $SC + N$, and its color to $CS + N$; making it appear.

ZAP: To shoot an enemy character, typically alien, with simulated lasers, beams, etc., in a computer game.

ZILCH: (U.S. slang) Zero. Nothing.

ZORK: To destroy an enemy character in a game program. Zorking is less impersonal than zapping.

ZYMURGY: From the Flatcat word game; applied chemistry for the purpose of brewing beer.

ALL OF THE PROGRAMS IN THIS BOOK ARE
AVAILABLE ON TAPE directly from the author.

Any 1 Program . . . \$10.00 postpaid

Any 2 Programs . . . \$18.00

Any 3 Programs . . . \$25.00

Any 4 Programs . . . \$30.00

California residents please add 6-1/2% sales tax.

Foreign Orders:

Canada, Mexico, Caribbean, and Central America add \$2.00

Europe, Asia, Africa, and South America add \$3.00

Send check or money order payable in U.S. currency to:

LARRY HATCH SOFTWARE LOFT
600 MENLO AVENUE
MENLO PARK, CALIFORNIA 94025

If postal remittance is nearly sufficient, the software will be
sent Air Mail.

If you own a Commodore 64®, you've got a full range of special computer features at your disposal—features like character, sprite, and full-screen dot resolution graphics, a full-fledged sound synthesizer chip, and more. This exciting new book takes full advantage of these outstanding features and gives you 25 wonderful programs that truly challenge and entertain you. You'll delight in the wide range of the games—from strategy to cryptography, mathematics to torpedoes, word recognition to gambling systems—a full circle of fun and amusement.

Let your imagination lead you through programs such as

Raging Robots • Crab Races • Flatcat • Crosstracks • Crypto Keno
• Saucers • Magic Cube • Magic Circles • 3-D Tic Tac Toe
• Gypsy Moth Invasion • Polyform • Reds • Quantum
Billiards Pocket Puzzle • Fill the Curve • Bandit • and more.

You'll agree that 25 ADVANCED GAMES FOR THE COMMODORE 64® gives you two dozen and one original, quality games that emphasize your fun.

Commodore 64® is a registered trademark of Commodore Business Machines.

A RESTON COMPUTER GROUP BOOK
RESTON PUBLISHING COMPANY, INC.

0-8359-7893-1